Univerza v Ljubljani
Fakulteta za računalništvo in informatiko
Fakulteta za matematiko in fiziko

Tinkara Toš

Algoritmi nad grafi v jeziku linearne algebre

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN MATEMATIKA

Mentor: doc. dr. Tomaž Dobravec

Ljubljana 2012

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00008/2012

Datum: 10.04.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko ter Fakulteta za matematiko in fiziko izdaja naslednjo nalogo:

Kandidat: **TINKARA TOŠ**

Naslov: **ALGORITMI NAD GRAFI V JEZIKU LINEARNE ALGEBRE**
GRAPH ALGORITHMS IN THE LANGUAGE OF LINEAR ALGEBRA

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

Poleg klasične predstavitve grafov z množicama vozlišč in povezav, lahko grafe predstavimo tudi na druge načine. Z algebraičnega vidika je še posebej zanimiva predstavitev grafa s pomočjo matrike, saj lahko nad grafi, predstavljenimi v taki obliki, izvajamo uveljavljene in dobro raziskane algebraične operacije.

V diplomskem delu preučite možne zapise grafa s pomočjo matrike. Preglejte različne načine zapisa redke matrike v računalniku (CRS, MSR, diagonalna matrika, ...) in natančno opišite njihovo strukturo. Preglejte tudi nekatere standardne algoritme nad grafi (iskanje v globino, Bellman-Fordov, Floyd-Warshallov, Primov algoritem, ...) in predstavite njihovo algebraično implementacijo. Vse naštete implementacije redkih matrik ter predstavljenih algoritmov sprogramirajte v programskem jeziku java. S pomočjo testov nad različnimi vhodnimi podatki ugotovite primernost posamezne implementacije za vse opisane algoritme.

Mentor:

doc. dr. Tomaž Dobravec

Dekan Fakultete za računalništvo in informatiko:

prof. dr. Nikolaj Zimic

Dekan Fakultete za matematiko in fiziko:

akad. prof. dr. Franc Forstnerič

Rezultati diplomskega dela so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

original izdane teme diplomskega dela

Izjava o avtorstvu diplomskega dela

Spodaj podpisana Tinkara Toš, z vpisno številko **63090185**, sem avtorica diplomskega dela z naslovom:

Algoritmi nad grafi v jeziku linearne algebre

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelala samostojno pod mentorstvom doc. dr. Tomaža Dobravca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 17. avgusta 2012 Podpis avtorice:

Kazalo

Povzetek

Abstract

1 Predstavitev **1**

1.1 Motivacija, cilji 1

2 Definicije in zapisi **3**

2.1 Definicije 3

2.2 Zapisi 8

3 Redke matrike **11**

3.1 Definicija in opis 11

3.2 Teorija grafov 12

3.3 Shranjevanje redkih matrik 15

3.4 Operacije na redkih matrikah 21

3.5 Cilj uporabe redkih matrik 24

4 Linearna algebra matrik in vektorjev **25**

4.1 Vektorji v prostoru 25

4.2 Matrike 27

5 Algoritmi **31**

5.1 BFS - iskanje v širino 31

5.2 Algoritem minimalne poti 34

5.3 Bellman-Fordov algoritem . . . 41
5.4 Floyd-Warshallow algoritem . . . 51
5.5 Primov algoritem . . . 56

6 Testiranje algoritmov **63**
6.1 Uvod v testiranje . . . 63
6.2 Množenje matrike z vektorjem . . . 64
6.3 Iskanje v širino (BFS algoritem) . . . 68
6.4 Iskanje minimalnih poti . . . 73
6.5 Bellman-Fordov algoritem . . . 77
6.6 Floyd-Warshallov algoritem . . . 84
6.7 Primov algoritem . . . 88

7 Zaključek **95**

Povzetek

Diplomsko delo predstavlja uporabnost dualnosti med grafom in njegovo sosedno matriko. V teoretičnem delu predstavi osnove teorije grafov in matrične algebre, predvsem se osredotoči na redke matrike in možnosti zapisov le-teh, ki upoštevajo število neničelnih elementov matrike. V delu so predstavljene operacije na redkih matrikah, preko katerih se naveže algoritme, ki v osnovi delujejo na grafih, a jih s pomočjo dualnosti grafa in matrike lahko spremenimo v zaporedje operacij na matrikah.
V nadaljevanju delo podrobneje predstavi nekaj algoritmov na grafih ter njihove komplementarne interpretacije na matrikah, ki v nadaljevanju postanejo predmet raziskovanja. Opisana je implementacija teh algoritmov v programskem jeziku Java in v nadaljevanju testiranje delovanja algoritmov na matrikah.
Pri praktičnem delu je poudarek na primerjavi med delovanjem teh algoritmov, ki uporabljajo standardni zapisa matrike s poljem, ter različico, ki uporablja zapis redke matrike. V delu je raziskano tudi, kdaj je katera različica boljša oziroma slabša.

Ključne besede
Matrika, sosedna matrika, matrika cene, redka matrika, teorija grafov, iskanje v širino, minimalne poti, Bellman-Fordov algoritem, Floyd-Warshallov algoritem, minimalno vpeto drevo, Primov algoritem.

Abstract

The thesis presents usefulness of duality between graph and his adjacency matrix. The teoretical part provides the basis of graph theory and matrix algebra mainly focusing on sparse matrices and options of their presentation witch takes into account the number of nonzero elements in the matrix. The thesis includes presentation of possible operations on sparse matrices and algorithms that basically work on graphs, but with help of duality between graph and his adjacency matrix can be presented with sequence of operations on matrices.
Practical part presents implementation of some algorithms that can work both with graphs or their adjacency matrices in programming language Java and testing algorithms that work with matrices.
It focuses on comparison in efficiency of algorithm working with matrix written in standard mode and with matrix written in format for sparse matrices. It also studies witch presentation of matrices works beter for witch algorithm.

Keywords
Matrix, adjacency matrix, sparse matrix, grapg theory, breath first search, minimum path, Bellman-Ford algorithm, Floyd-Warshall algorithm, minimum spanning tree, Prim's algorithm.

Poglavje 1

Predstavitev

1.1 Motivacija, cilji

Dualnost med kanonično predstavitvijo grafov kot abstraktno kolekcijo vozlišč in povezav ter redko sosedno matriko je del teorije grafov vse od njenega začetka, matrična algebra pa je pri predstavitvi te dualnosti lahko zelo uporaben pripomoček. Vseeno se matrike tradicionalno niso uporabljale za praktično računanje z grafi, predvsem zato, ker predstavitev matrike kot 2-dimenzionalna tabela ni najučinkovitejša predstavitev redkega grafa. Z razvojem in večanjem učinkovitosti podatkovnih struktur in algoritmov za redka polja in matrike je postalo mogoče razviti praktičen pristop, povezan s polji za izvajanje operacij na velikih redkih matrikah.

Vsak problem na matrikah je problem na grafih in vsak problem na grafih je problem na matrikah.

Ta dualnost je osnovni koncept pri algoritmih, ki delujejo na grafih, prestavljenih z matrikami.

Ker so matrike postale zelo primeren način za predstavitev grafov, so na primer v MATLABU matrike postale najpogosteje uporabljena struktura za reševanje problemov, povezanih z grafi. Veliko raziskovalnih skupin gradi knjižnice za paralelne algoritme na grafih s pomočjo redkih matrik.

Naša naloga v tem diplomskem delu bo preučiti nekaj algoritmov na grafih, pred-

stavljenih z matrikami, ter možne načine zapisa redkih matrik in gostih matrik ter operacij na le-teh. Ugotovili bomo, kateri algoritmi, operacije in zapisi matrik so časovno najmanj potratni.
Predvidevam, da bo na učinkovitost algoritmov poleg velikosti matrik vplivala tudi uporaba zapisa redke matrike.
Naš cilj bo ugotoviti, ne samo kateri zapis redke matrike je manj časovno potraten, ampak tudi kdaj, saj predvidevam, da se bodo na različnih matrikah različni zapisi matrik obnašali različno.

Poglavje 2

Definicije in zapisi

2.1 Definicije

Definicija *KOLOBAR* je množica K z dvema računskima operacijama, ki ju zaradi preprostosti imenujemo seštevanje in množenje, ni pa nujno, da pomenita klasično seštevanje in množenje. Označimo ju z znakoma + (plus) in * (krat). Za poljubne elemente kolobarja a,b,c mora veljati [9] :

- $a + b = b + a$

  (komutativnost za seštevanje)

- $a + (b + c) = (a + b) + c$

  (asociativnost za seštevanje)

- obstaja **nevtralni element za seštevanje** (označimo z oznako 0) , da velja $a + 0 = 0 + a = a$

- poljubni element a ima **nasprotni element** -a, tako da velja $a + (-a) = (-a) + a = 0$

- $a * (b * c) = (a * b) * c$

  (asociativnost za množenje)

- distributivnost iz leve in desne strani, ki povezuje seštevanje in množenje

$$a * (b + c) = a * b + a * c$$

$$(a + b) * c = a * c + b * c$$

Kolobar je za operacijo + Abelova grupa.
Kolobar določajo tile podatki: K množica njegovih elementov, operacija "+" imenovana seštevanje in "*" imenovana množenje. Če enega izmed teh podatkov spremenimo, dobimo drug kolobar [3].

Definicija *OBSEG* je v abstraktni algebri ime za algebrsko strukturo, v kateri je možno brez omejitev seštevati, odštevati, množiti in deliti (razen deljenja z 0).
Obseg je množica O z dvema računskima operacijama, ki ju zaradi preprostosti imenujemo seštevanje in množenje in ju označimo z znakoma + (plus) in * (krat), lahko pa ti operaciji pomenita kaj drugega kot navadno seštevanje in množenje števil.
Da bo algebrska struktura res obseg, mora veljati:

- (O, + ,*) je kolobar
- obstaja nevtralni element za množenje, ki ga označimo z 1 (enota) in je različen od nevtralnega elementa za seštevanje (0)

  $1 * a = a * 1 = a$

- za vsak od 0 različen element a obstaja inverzni element a^{-1}, tako da velja:

  $a * a^{-1} = a^{-1} * a = 1$

Če v nekem obsegu velja še komutativnost za množenje ($a * b = b * a$), potem je to **komutativni obseg** [9].

Definicija Naj bo V množica in F obseg (na primer obseg realnih ali kompleksnih števil) in naj bosta definirani naslednji operaciji:
- operacija vektorske vsote ali seštevanja vektorjev, označena kot $v + w$ (kjer sta $v, w \in V$)

- operacija množenja s skalarjem, označena z $a * v$ (kjer je $v \in V$ in $a \in F$)
Množico V tedaj po definiciji imenujemo *VEKTORSKI PROSTOR* nad obsegom F, če veljajo naslednje značilnosti [9] :

- če $v, w \in V$, potem tudi $v + w \in V$

  (z drugimi besedami: V je zaprta za seštevanje vektorjev)

- $u + (v + w) = (u + v) + w$

  (asociativnost seštevanja vektorjev v V)

- v množici V obstaja **nevtralni element 0**, da za vse elemente $v \in V$ velja $v + 0 = v$

  (obstoj aditivne identitete v V)

- za vsak $v \in V$ obstaja element $w \in V$, da je $v + w = 0$

  (obstoj nasprotnih vrednosti v V)

- $v + w = w + v$

  (komutativnost vektorske vsote v V)

- $a * v \in V$

  (zaprtost V za množenje s skalarjem)

- $a * (b * v) = (ab) * v$

  (asociativnost množenja s skalarjem v V)

- Če 1 označuje identiteto za množenje v obsegu F, potem velja $1 * v = v$

  (nevtralnost elementa 1)

- $a * (v + w) = a * v + a * w$

  (distributivnost glede na seštevanje vektorjev)

- $(a + b) * v = a * v + b * v$

  (distributivnost glede na seštevanje v obsegu)

Definicija *MATRIKA* je v matematiki pravokotna razpredelnica števil ali v splošnem elementov poljubnega kolobarja. V tem delu bodo elementi realna števila, če ni drugače rečeno. Vodoravni "črti" matrike rečemo vrstica, navpični pa stolpec. Matrika z m vrsticami in n stolpci se imenuje m x n matrika, m in n pa sta njeni razsežnosti. Element, ki leži v i-ti vrstici in j-tem stopcu matrike (kjer vrstice in stolpce ponavadi štejemo od 1 naprej, razen v kakšen programskem jeziku od 0 naprej), je (i,j)- ti element, v delu bomo za matriko A označili (i,j)-ti element z $a_{i,j}$. Matriko A razsežnosti m x n lahko torej definiramo s predpisom

$$A := (a_{i,j})_{mxn} \text{ [3]}.$$

Definicija *REDKA MATRIKA* je matrika, ki ima več ničelnih elementov kot pa neničelnih.

Definicija *GRAF* G je neprazna množica elementov, ki jih imenujemo **vozlišča grafa**, in množica (neurejenih) parov vozlišč, ki jih imenujemo **povezave grafa**. Množico vozlišč označimo z **V(G)** ali pa kar z V, množico povezav pa **E(G)** oziroma krajše kar E. Če sta v in w točki grafa G, potem za povezavi vw oziroma wv rečemo, da povezujeta točki v in w [8].

UTEŽENI GRAF je definiran kot množica vozlišč V= $v_1, v_2, ...v_n$, množico povezav E= $(v_i, v_j : w_{ij})$; i,j =1,2,...n, pri čemer z $(v_i, v_j : w_{ij})$ označimo povezavo, ki povezuje vozlišče v_i z vozliščem v_j, z utežmi w_{ij}, ter preslikavo W iz E v $\mathbb{R}$ uteženi graf pa kot G(V,E,W). Uteženi graf je lahko usmerjen ali neusmerjen, cikličen ali necikličen [12].

USMERJEN GRAF je tisti graf, ki ima vse povezave usmerjene. Iz enega vozlišča se lahko premaknemo v drugega (tako da sledimo smeri povezave), nazaj pa po isti poti ne moremo, razen če je le-ta dvosmerna [12] .

NEUSMERJEN GRAF je tisti graf, ki nima usmerjenih povezav, če med vozliščema obstaja povezava, lahko gremo iz prvega v drugo vozlišče in tudi obratno.

CIKLIČNI GRAF je graf, ki ga sestavlja samo en cikel oziroma so vozlišča med seboj povezana v zaprto verigo. Pri takem grafu je število vozlišč enako številu

povezav in vsako vozlišče ima stopnjo 2. Označujemo ga z C_n, kjer je n število vozlišč [9].
NECIKLIČNI GRAF je graf, pri katerem povezave ne sestavljajo nobenega cikla (na sprehodu iz katerikoli točke se nam ne more zgoditi, da bi se vrnili v isto točko, če se ne vrnemo po isti poti nazaj) [9] .
POVEZAN GRAF je graf, katerega vozlišč ne moremo razporediti v 2 skupini tako, da med eno in drugo skupino ne bi obstajala povezava, ki bi ju povezovala.

Definicija *STOPNJA VOZLIŠČA* je število povezav, ki so vezane na vozlišče. Pri tem se zanke štejejo dvakrat. Stopnjo oglišča označujemo z deg(v).

Definicija *ZANKA* je cikel dolžine 0 oziroma pot iz vozlišča v vozlišče samo.

Definicija *DREVO* je v teoriji grafov graf, v katerem sta poljubni dve vozlišči povezani z natanko eno enostavno potjo. Drevo je v bistvu vsak povezan graf brez ciklov. **Gozd** je nepovezana unija dreves [9].

Definicija *POVEZANOST VOZLIŠČ*
V neusmerjenem grafu sta vozlišči u in v **povezani**, če obstaja med njima neka pot, kar pomeni, da lahko nekako pridemo po grafu od ene do druge.
V usmerjenem grafu sta vozlišči u in v [14]

- **močno povezani** natanko takrat, ko obstaja neka pot od u do v, in tudi neka pot od v do u upoštevajoč smeri povezav;
- **šibko povezani** natanko takrat, ko obstaja neka pot med u in v, če zanemarimo smeri povezav.

(Krepko, šibko) **povezana komponenta** je množica vozlišč, ki so vse paroma (krepko, šibko) med seboj povezane in v katero ne moremo dati nobene nove točke, ne da bi prejšnji pogoj nehal veljati [14] .

Definicija *NEODVISNA MNOŽICA* je neusmerjen graf oziroma množica vozlišč grafa, ki so paroma neodvisne. To pomeni, da noben par vozlišč iz te množice ni soseden. Neodvisna množica je **največja**, če ni poddmnožica katerikoli druge neodvisne množice.

2.2 Zapisi

MNOŽENJE

Množenje med vektorjem in matriko ali pa med dvema matrikama bomo včasih zapisali kar z Matlab zapisom

$$A\ op_1.op_2\ v.$$

Namesto vektorja (v) bi seveda lahko bila matrika (B).

Za osnovno matrično množenje bi torej veljalo $op_1 = +$ in $op_2 = *$

Ta zapis nam bo močno olajšal zapis nekaterih algoritmov.

Če na primer namesto kolobarja z operacijama + (plus) in * (krat) uporabimo produkt v tropski algebri, kjer namesto plusa uporabimo max, namesto krat pa +, dobimo za produkt zapis $A\ max. +\ v$. Še bolj prav nam bo prišel produkt $A\ min. +\ v$, a o tem kasneje.

GRAF

Grafe bomo zapisovali kot G=(V,E), kjer bo V množica N vozlišč grafa in E množica M povezav grafa. Drug možen zapis grafa bo z matriko A, kjer je A matrika dimenzije N x N in ima M neničelnih elementov oziroma A(i,j) (element matrike v i-ti vrstici in j-tem stolpcu) bo različen od 0, kadar bo med vozliščema i in j obstajala povezava. Vrednost tega elementa je odvisna od tega, kaj vse želimo z matriko na grafu predstaviti (cene povezav, obstoj povezave), zato bo pri vsakem primeru posebej označeno, kaj natanko predstavljajo elementi matrike.

MATRIKE

Matrike lahko vsebujejo logične vrednosti $\mathbb{B}$ celoštevilske $\mathbb{Z}$ ali realne $\mathbb{R}$.

Zapis na primer A : $\mathbb{R}^{5x6x7}$ pove, da je A 3-dimenzionalno polje z 210 realnimi elementi dimenzije $5 \times 6 \times 7$.

Indeksiranje A(i,j) uporabimo tudi na vektorju, v(i) pomeni i-ti element vektorja.

Indeksiranje pa lahko uporabimo tudi na izrazih,f na primer $[(I - A)^{-1}](i, j)$ je zapis (i,j) v inverzu matrike I-A.

Deli matrik bodo zapisani z Matlab zapisom. Na primer A(1:5;[3 1 4 1]) je matrika dimenzije 5 x 4, ki vsebuje elemente prvih petih vrstic iz stolpcev 3 1 4 1 v tem vrstnem redu.

Če je I indeks ali pa množica indeksov, je $A(I; :)$ podmatrika iz A z vrsticami, ki so označene v I in vsemi stolpci (znak $:$ pomeni vsi možni)

DIMENZIJA	IME	PRIMER
0	skalar	s
1	vektor	v
2	matrika	A

Tabela 2.1: Tabela algebrskih struktur in zapisov

Poglavje 3

Redke matrike

3.1 Definicija in opis

Definicija *Redka matrika* je matrika, ki ima več ničelnih elementov kot pa neničelnih.
Redke matrike lahko razdelimo na 2 vrsti, in sicer na strukturirane in nestrukturirane.

- **STRUKTURIRANE** so tiste, pri katerih neničelni zapisi ustvarjajo nek vzorec, največkrat v obliki nekega števila diagonal. Neničelni zapisi lahko ležijo v blokih (podmatrike) istih velikosti.
- **NESTRUKTURIRANE** so tiste, pri katerih nimamo pravilne postavitve neničelnih elementov.

Večina tehnik je narejenih za nestrukturirane redke matrike, tako da razlike med tema dvema tipoma vsaj v preteklosti niso kazale na to, da bi zaradi strukturiranosti ali nestrukturiranosti ti algoritmi bolje oziroma hitreje delovali. Močno vlogo pa lahko ima ta lastnost redkih matrik pri iterativnih rešitvah problemov, kjer je ena pomembnejših operacij množenje matrike z vektorjem. Učinkovitost te operacije je lahko močno odvisna od tega ali je matrika strukturirana ali ne. Na vektorskih računalnikih je naprimer shranjevanje matrik po diagonalah idealno, bolj splošne sheme pa lahko trpijo, ker zahtevajo indirektno naslavljanje [5] .

3.2 Teorija grafov

Graf je odličen za prikaz relacij med nekimi točkami, na primer naše točke so lahko mesta; povezava med točko A in B pa pomeni, da med njima obstaja prevozna cesta. Če pa želimo pri računanju z grafi uporabiti še linearno algebro, moramo graf znati predstaviti z matriko, kar lahko storimo na več možnih načinov.
Ta dualnost omogoči algoritmom na grafu, da jih prestavimo kot zaporedje linearnih algebraičnih operacij. Z matriko lahko predstavimo več lastnosti grafa, v večini primerov pa graf predstavimo s trojicami (u,v,w), kar predstavlja usmerjeno povezavo s ceno w iz vozlišča u do v. Ta povezava se odraža v neničelni vrednosti w na lokaciji (u,v) v naši matriki.
Prednosti zapisa grafa z matriko [1]

- Sintaktična kompleksnost

  (Veliko algoritmov na grafih postane z linearno algebro in predstavitvijo z matrikami bolj kompaktno in lažje za razumevanje.)

- Enostavna implementacija

  (S pomočjo matrik lahko algoritmi na grafih izkoriščajo programsko infrastrukturo za paralelno računanje.)

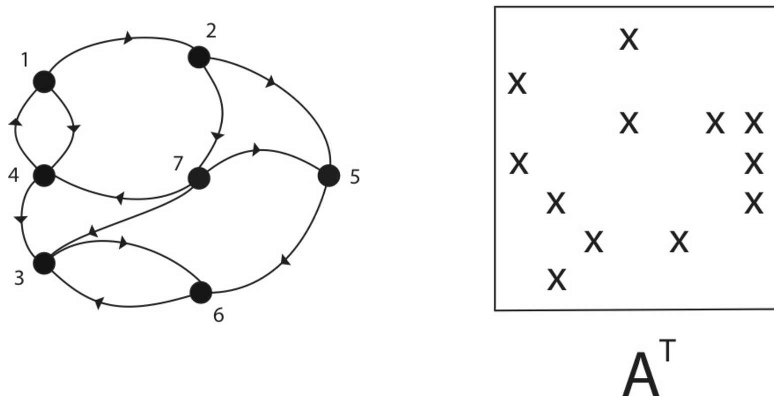
- Zmogljivost

  (Ker ti algoritmi bolj poudarjajo učinkovit podatkovni dostop, jih je lažje optimizirati.)

Slika 3.1 prikazuje splošen primer dualnosti med grafom in matriko; glede na to, kaj so X-i na sliki, pa ločimo več različnih matrik.

3.2.1 Matrika sosednosti

Usmerjen graf G=(V,E) lahko predstavimo z matriko sosednosti (ang. **adjacency matrix**) A, za katero velja, da $a_{i,j} = 1$ natanko takrat, ko v originalnem grafu obstaja povezava med vozliščem i in vozliščem j [1]. Glede na to, da matrika predstavlja **sosede** vozlišča, moramo definirati, ali je vozlišče sosed samemu sebi.

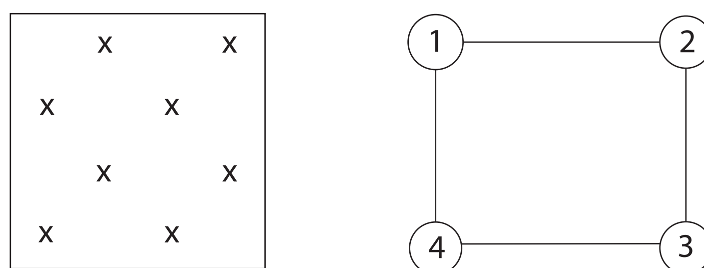


Slika 3.1: Primer dualnosti med grafom in matriko

Rekli bomo, da vozlišče ni samemu sebi sosed, saj iz vozlišča i do samega sebe ne moremo v natanko 1 koraku, do sebe "pridemo" v 0 korakih ali pa v najmanj 2. Zato definiramo $a_{i,i} = 0$.

i-ta vrstica v matriki A nam pove, v katera vozlišča lahko pridemo iz vozlišča i v 1 koraku. Če imamo v i-ti vrstici j-ti stolpec enak 0, to pomeni, da iz i ne moremo do j v enem koraku.

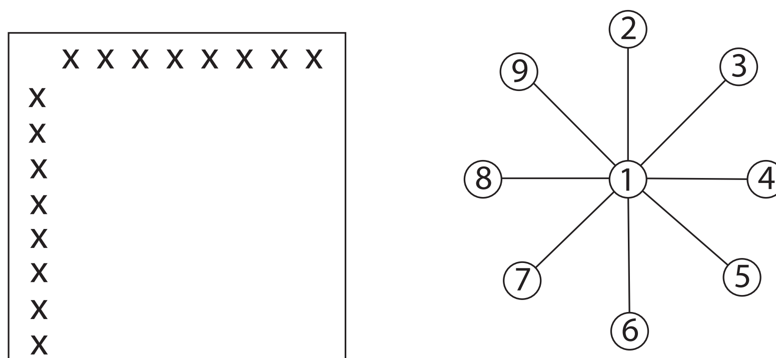
V primeru, ko so povezave obojestranske (če lahko pridemo od A do B in tudi od B do A), dobimo simetrično matriko.



Slika 3.2: Primer matrike sosednosti za 4 točke

Transponirana matrika sosednosti

Transponirati matriko pomeni, da i-ti stolpec matrike A postane i-ta vrstica matrike A^T (transponirana matrika matrike A).



Slika 3.3: Primer matrike sosednosti za več točk

Pri transponirani matriki sosednosti i-ta vrstica pove, iz katerih vozlišč pridemo v vozlišče i v 1 koraku.

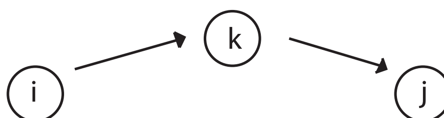
Uporaba matrike sosednosti

S potenciranjem matrike sosednosti dobimo matriko, katere elementi predstavljajo število poti med dvema vozliščema v toliko korakih, kakršna je potenca (npr. s četrto potenco matrike A dobimo informacijo o številu poti med poljubnima 2 vozliščema v 4 korakih).

Trditev 3.2.1 *A^r_{ij} je število poti med i in j dolžine r.*

Dokaz (z indukcijo)

- za r=1 trditev drži po definiciji
- $r-1 \longmapsto: r : a^r_{ij} = \sum_{k=1}^{n} a^{r-1}_{ik} a_{kj}$



Pomoč s sliko: Seštejemo vse a^{r-1}_{ik} pri katerih je a_{kj} različna od 0, dobimo skupno število vseh poti med i in j dolžine r [4].

3.2.2 Matrika dosegljivosti

A je matrika sosednosti, matriko G:= A + I imenujemo matrika dosegljivosti. Če je element matrike G $g_{ij} \neq 0$, to pomeni, da lahko iz vozlišča i do vozlišča j pridemo v NAJVEČ enem koraku.

Trditev 3.2.2 *r-ta potenca matrike G v kolobarju z operacijama ALI in IN predstavlja dosegljivost v r korakih.*
$G^{\diamond r} \neq 0 \Leftrightarrow$ *iz vozlišča i lahko pridem v vozlišče j v največ r korakih*

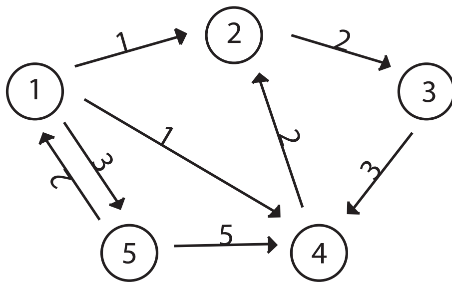
Dokaz (z indukcijo)
$G^{\diamond r}(i,j)$ je produkt i-te vrstice $G^{\diamond r-1}(i,j)$ in j-tega stolpca G. $G^{\diamond r}(i,j) = OR_k(G^{\diamond r-1}(i,k)$ AND G(k,j)). Torej $G^{\diamond r}(i,j) = 1$, če lahko pridem iz i v k v r korakih in iz k v j v enem koraku za katerikoli k [4].

3.2.3 Matrika cen

Utežen graf G=(V,E,w) predstavimo z matriko cen C, kjer za element velja:

$$c_{ij} = \begin{cases} 0; & i = j \\ w_{ij}; & \text{če med i in j obstaja povezava} \\ \infty; & sicer \end{cases}$$

Matrika C oziroma element c_{ij} nam pove, koliko nas stane, da pridemo iz vozlišča i v j. S to matriko si lahko pomagamo predvsem pri problemih iskanja minimalnih poti.



$$\begin{pmatrix} 0 & 1 & \infty & 1 & 3 \\ \infty & 0 & 2 & \infty & \infty \\ \infty & \infty & 0 & 3 & \infty \\ \infty & 2 & \infty & 0 & \infty \\ 2 & \infty & \infty & 5 & 0 \end{pmatrix}$$

3.3 Shranjevanje redkih matrik

Pri operacijah na grafih, ki so predstavljeni z matrikami, si v največji meri pomagamo z redkimi matrikami, zato sta seveda uspešnost in časovna zahtevnost takih

algoritmov odvisna od implementacije in shranjevanja redkih matrik. Naš osnovni cilj pri redkih matrikah je učinkovito shranjevanje, saj ima malo neničelnih elementov. Naša ideja temelji na tem, da shranjujemo le neničelne elemente in ne cele matrike s pripadajočim velikim številom ničel. Želimo, da bi velikost porabljenega prostora bila proporcionalna s številom neničelnih elementov in ne z velikostjo cele matrike.
Primeri bodo prikazani na matriki:

$$A = \begin{pmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{pmatrix}$$

3.3.1 CSR ZAPIS (Compressed Storage by Rows)

CSR zapis (Compressed Storage by Rows) shranjuje neničelne elemente, ki so urejeni v odvisnosti od svoje pozicije v matriki po vrsticah, torej elementi so shranjeni po vrsti, po vrsticah in v vsaki vrstici po stolpcih od leve proti desni.
Shraniti moramo 3 polja data, ind in poin. Data je polje z realnimi elementi matrike, ki so urejeni po vrsti po vrsticah kot ležijo v matriki. Ind vsebuje naravna števila, ki predstavljajo indekse stolpcev elementov. Data in ind sta povezana tako, da ima element i iz data shranjen svoj indeks stolpca v i-tem elementu polja ind. (če $data[k] = a_{ij}$, potem je ind[k]=j) Poin pa shranjuje indeks elementa v data, s katerim se začne posamezna vrstica, npr na prvem mestu polja poin bo indeks elementa v data, s katerim se začne prva vrstica matrike. Poin je vedno dolžine n+1, kjer je n dimenzija matrike, za n+1 polje določimo nnz + 1, kjer je nnz število neničelnih elementov v matriki.
Prihranek prostora pri shranjevanju v tem zapisu je precej opazen, saj namesto n^2 polj potrebujemo le 2nnz + n + 1 polj.
Pri simetričnih matrikah lahko še bolj prihranimo, saj lahko shranjujemo le zgornji trikotnik matrike, težava je le v tem, da s takim zapisom matrike težko računamo oziroma so algoritmi veliko bolj kompleksni.

CSR zapis je precej splošen, ne potrebujemo nobenih predpostavk o redkosti matrike, da ga lahko uporabimo, in ne shranjujejo nobenih nepotrebnih polj; njegova pomanjkljivost je, da potrebujemo indirektno naslavljanje polj na skoraj vsakem koraku uporabe [5] [1].

data	1 2 3 4 5 6 7 8 9 10 11 12
ind	1 4 1 2 4 1 3 4 5 3 4 5
poin	1 3 6 10 12 13

Slika 3.4: Primer CSR zapisa na matriki A

3.3.2 CSC ZAPIS (Compressed Storage by Columns)

CSC zapis je variacija CSR zapisa . V bistvu je CSC zapis CSR zapis, ki namesto z vrsticami vrstni red elementov določa s stolpci. Lahko bi rekli, da je CSC CSR zapis za A^T. V data shranjujemo vrednosti matrike po stolpcih od levega proti desnemu, v vsakem stolpcu pa od zgoraj navzdol. Ind shranjuje zaporedno številko vrstice, v kateri se element nahaja. Povezava med data in ind ostaja enaka kot pri CSR zapisu. Poin shranjuje indeks elementa v data, s katerim se začne posamezen stolpec, namesto vrstice, kot pri CSR [5] [1] .

data	1 2 3 4 5 6 7 8 9 10 11 12
ind	1 1 2 2 2 3 3 3 3 4 4 5
poin	1 4 7 2 9 13

Slika 3.5: Primer CSC zapisa na matriki A

3.3.3 ZAPIS KOORDINATNEGA SISTEMA (ZAPIS S TERKAMI)

Manjkrat uporabljen je zapis s terkami, kjer preprosto shranjujemo terke oblike (podatek, vrstica, stolpec).

V ta namen moramo shraniti 3 polja data, row in col, kjer je data polje z realnimi elementi matrike, row vsebuje naravna števila, ki predstavljajo indekse vrstic elementov v matriki, col pa vsebuje naravna števila, ki predstavjajo indekse stolpcev elementov v matriki.

Polja so povezana tako, da imamo za k-ti element v polju data na k-tem mestu v row shranjen indeks vrstice tega elementa v matriki in na k-tem elementu v col indeks stolpca elementa v matriki [5] [1].

Prednosti: enostavnost in fleksibilnost.

data	1	2	3	4	5	6	7	8	9	10	11	12
row	1	1	2	2	2	3	3	3	3	4	4	5
col	1	4	1	2	4	1	3	4	5	3	4	5

Slika 3.6: Primer zapisa s terkami na matriki A

3.3.4 MSR ZAPIS (Modified Sparse Row)

MSR zapis izkorišča dejstvo, da ima večina matrik običajno neničelno diagonalo (veliko algoritmov to celo zahteva) in/ali do te diagonale dostopamo pogosteje kot do drugih elementov v matriki. Zato je uporabno, da jo hranimo posebej oziroma prej.

V ta namen pri tem zapisu shranjujemo le 2 polji, data in ind, data z realnimi števili in ind z naravnimi. V polju data so na prvih n mestih vrednosti elementov na diagonali urejene po vrsti, n+1 pozicija ponavadi ni zasedena, lahko pa to polje nosi kakšno drugo informacijo o matriki; s pozicijo n+2 pa se začnejo ostali

neničelni elementi matrike, ki jih shranjujemo po vrsti po vrsticah, podobno kot pri CSR, le da spuščamo elemente na diagonali, saj so zapisani že na začetku. Za prvih n elementov polja data torej velja, da na i-ti poziciji leži element v matriki, ki je v i-tem stolpcu in i-ti vrstici.

Ind ima naslednjo strukturo:

- v prvih n poljih imamo shranjen indeks elementa matrike, s katerim se začne vrstica, pri čemer ignoriramo diagonalne elemente. Če je diagonalni element edini element v neki vrstici matrike, tja vstavimo indeks, ki ne obstaja v tabeli data.
- n+1 element je indeks, ki ne obstaja v tabeli data (lahko kar za ena povečan maksimalen indeks v data).
- Od n+2 elementa v ind naprej velja, da za k-ti element v data (kjer je k večji ali enak n+2) hranimo indeks stolpca, v katerem se ta element nahaja v matriki.

Poraba prostora je malce manjša kot pri zgornjih zapisih, zapis pa je najbolj uporaben, kadar potrebujemo hitro identificiranje elementov na diagonali [10] .

data	1	4	7	11	12	*	2	3	5	6	8	9	10
ind	7	9	10	13	14	14	4	1	4	1	4	5	3

Slika 3.7: Primer MSR zapisa na matriki A

Zvezdica prikazuje neuporabljen element.

Opazimo, da ind(n)= ind(n+1)=14, kar pomeni, da je zadnja vrstica vrstica ničel, če bi umaknili diagonalne elemente (edini element v tej vrstici je diagonalni element).

3.3.5 SHRANJEVANJE DIAGONALNIH MATRIK

Definicija *Diagonalna matrika* je matrika, ki ima neničelne matrike samo na majhnem številu diagonal.

Te diagonale lahko shranimo v pravokotnem polju DIAG(1:n,1:Nd), kjer je Nd število diagonal. Oddaljenost diagonale od glavne diagonale mora seveda biti znana. Te oddaljenosti shranimo v polju IOFF(1:Nd). Torej je element $a_{i,i+ioff(j)}$ originalne matrike na poziciji (i,j) polja DIAG
DIAG (i,j) $\leftarrow a_{i,i+ioff(j)}$
Vrsti red shranjevanja diagonal ni pomemben, razen če do katere od diagonal dostopamo večkrat, potem bomo v prednosti, če te diagonale shranimo na začetek. Opazimo tudi, da so vse diagonale krajše od glavne diagonale, zato nekateri elementi DIAG ne bodo uporabljeni [5] .

$$B = \begin{pmatrix} 1 & 0 & 2 & 0 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 0 & 6 & 7 & 0 & 8 \\ 0 & 0 & 9 & 10 & 0 \\ 0 & 0 & 0 & 11 & 12 \end{pmatrix}$$

DIAG=

*	1	2
3	4	5
6	7	8
9	10	*
11	12	*

IOF=

-1	0	2

Slika 3.8: Primer shranjevanja diagonalne matrike na matriki B

Splošnejša rešitev, uporabljena na vektorskih računalnikih, je Ellpack-Itpack zapis. Predpostavka, ki jo moramo privzeti, je, da imamo največ Nd neničelnih elementov v vrstici, kjer je Nd majhen. Nato potrebujemo 2 polji velikosti n x Nd

COEF in JCOEF. Neničelne elemente vsake vrstice matrike shranimo v vrstico COEF(1:n,1:Nd), pri čemer zapolnimo vrstice z ničlami, če je potrebno. V polje JCOEF(1:n,1:Nd) shranimo pozicijo stolpca za vsak zapis v COEF [5].

$$
\text{COEF}=\begin{bmatrix} 1 & 2 & 0 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \\ 9 & 10 & 0 \\ 11 & 12 & 0 \end{bmatrix} \quad \text{JCOEF}=\begin{bmatrix} 1 & 3 & 1 \\ 1 & 2 & 4 \\ 2 & 3 & 5 \\ 3 & 4 & 4 \\ 4 & 5 & 5 \end{bmatrix}
$$

Slika 3.9: Primer shranjevanja diagonalne matrike z Ellpach-Itpackovim zapisom na matriki B

Določena številka stolpca mora biti izbrana za vsak ničelen element, ki je dodan, da zapolni vrstico v A. V tem primeru je to za vrstice 1, 4 in 5. Tukaj smo določili, da se v JCOEF zapiše kar številka vrstice, v kateri je ničla. Katero številko tu izberemo, ni pomembno, dobra bi bila vsaka med 1 in n, vseeno pa je dobro, da ne vstavljamo iste številke prepogosto, npr. vedno neko konstanto, saj nam lahko to poslabša učinkovitost nekaterih algoritmov.

3.4 Operacije na redkih matrikah

3.4.1 Množenje matrik

Za učinkovite algoritme potrebujemo tudi učinkovito matrično množenje redkih matrik. Neodvisno od načina shranjevanja je časovna zahtevnost za množenje matrike dimenzije NxN z vektorjem z M neničelnimi elementi približno $2M^2/N$. Z uporabo tega modela je tudi enostavno izračunati kompleksnost večine linearnih algebraičnih algoritmov na grafih.

3.4.2 Množenje matrike z vektorjem

Produkt med matriko in vektorjem (y= Ax) je zelo pomembna operacija, ki jo potrebujemo za večino iterativnih algoritmov za reševanje redkih linearnih sistemov. Implementacija je odvisna od načina shranjevanja matrike. Recimo, da imamo matriko shranjeno v CSR zapisu.
Operacijo v tem zapisu lahko izrazimo na običajem način : $y_i = \sum_j a_{i,j} x_j$, saj množenje prehaja čez vrstice matrike.

Algoritem 3.4.2: Množenje matrike v CSR zapisu z vektorjem [13]

```
for i:=1 to n step 1 {
        y(i)=0;
}
for j:=poin(i) to  poin(i+1)−1 step 1{
        y(i)=y(i)+data(j)*x(ind(j));
}
```

Rešitev dobimo v y.
Opazimo, da vsaka iteracija izračuna drugo komponento končnega rezultata (vektorja). To je prednost, saj lahko vsako komponento izračunamo neodvisno, torej lahko uporabimo paralelizem. Prav tako opazimo, da metoda množi le neničelne elemente, zato naredimo le 2nnz operacij, medtem ko jih pri običajnem množenju matrik potrebujemo kar $2n^2$ (n predstavlja dimenzijo matrike, nnz pa število neničelnih elementov v matriki).
Za produkt $y = A^T x$ ne moremo uporabiti direktne enačbe $y_i = \sum_j (A^T)_{i,j} x_j = \sum_j a_{j,i} x_j$, saj to pomeni, da bi morali prečkati stolpce matrike, kar pa je zelo neučinkovito, če imamo matriko shranjeno v CSR zapisu. Zato moramo upoštevati naslednje:

- namesto za vse j naredi za vse i;
- y_i postane $y_i + a_{j,i} x_j$.

Algoritem 3.4.4: Množenje transponirane matrike v CSR zapisu z vektorjem [6] [1]

```
for i:=1 to n step 1 {
        y(i)=0;
}
for i:=poin(j) to  poin(j+1)−1 step 1 {
        y(ind(i))=y(ind(i))+data(i)∗x(j);
}
```

MSR ZAPIS

Recimo, da imamo matriko shranjeno v MSR zapisu. Ta zapis ni ravno idealen za množenje matrike z vektorjem, saj pri tej operaciji potrebujemo podatek o stolpcu in vrstici elementa, s katerim trenutno množimo vektor, kar pa iz tega zapisa ni direktno razvidno.

Algoritem deluje tako, da gre po vseh elementih, od n+2 elementa naprej, ugotovi, v kateri vrstici matrike se nahaja ta element (stolpec je viden neposredno iz tabele ind) in nato zmnoži element z elementom v ustrezni vrstici vektorja in ga prišteje rezultatu v ustrezno vrstico. Na koncu prišteje še vrednosti, ki jih dobimo, če množimo diagonalo, tu ni večjih težav, saj imajo ti elementi isti indeks stolpca in vrstice, element, s katerim množimo (iz vektorja x), pa je tudi v isti vrstici.

Algoritem 3.4.6: Množenje matrike v MSR zapisu z vektorjem [7]

```
for j:=n+1 to end of data step 1 {
        for k:=1 to n step 1{
                if ((j < ind[k] AND j> ind[k+1])
                        OR j=ind[k]) {
                        vrstica=k;
                }
        }
        y(vrstica)=y(vrstica) + data(j)∗x(ind(j))
}
```

```
for i:=1 to  n step 1 {
        y(i)=y(i)+data(i)*x(i);
}
```

ZAPIS S TERKAMI

Recimo, da imamo matriko shranjeno v zapisu s terkami. Zapis je precej primeren za množenje matrike z vektorjem, saj je algoritem zelo preprost, enostavno gremo čez vse elemente v polju data in zmnožimo element z elementom v vektorju, ki leži na pravem mestu. Enostavno je ugotoviti, kako med seboj množiti elemente, saj imamo za vsak element v data neposredno dostopna tudi podatka o indeksu vrstice in stolpca.

Algoritem 3.4.8: Množenje matrike v zapisu s terkami z vektorjem [1]

```
for i:=1 to  end of data step 1{
        y(row(i))=y(row(i))+data(i)*x(col(i));
}
```

3.5 Cilj uporabe redkih matrik

- Prostor pri shranjevanju redke matrike bi moral biti sorazmeren s številom vrstic, številom stolpcev in številom neničelnih elementov.
- Časovna zahtevnost operacije na redki matriki bi morala biti sorazmerna z velikostjo podatkov, do katerih dostopamo, in s številom aritmetičnih operacij, izvedenih na neničelnih elementih.

Poglavje 4

Linearna algebra matrik in vektorjev

4.1 Vektorji v prostoru

Množici vseh n-terk realnih števil, ki jo označimo z $\mathbb{R}^n$, rečemo **n-prostor**.
Strukturi u=$(a_1, a_2...a_n)$ rečemo **vektor**.
Števila a_i so koordinate, komponente, vnosi oziroma kar elementi vektorja u. Kadar govorimo o prostoru $\mathbb{R}^n$, rečemo elementom (številom) iz $\mathbb{R}$ kar **skalarji**.
Vektorja u in v sta **enaka**, če imata enako število elementov in imata istoležeče elemente enake. Primer: (1,2,3) in (2,3,1) imata sicer enake elemente, a ne na istih mestih, zato nista enaka. [2]
Vektor (0,0,...,0), ki ima vse elemente enake 0, se imenuje **ničelni vektor** in ga ponavadi označimo kar z 0.
Včasih vektor v n-prostoru napišemo navpično, namesto horizontalno. Takšnemu vektorju rečemo **stolpični vektor**, obratno vektorjem, zapisanim horizontalno, rečemo **vrstični vektorji**.
Vse operacije, definirane na vrstičnih, so analogno definirane na stolpičnih vektorjih. [2]
OPERACIJE NA VEKTORJIH
Recimo, da imamo dva vektorja u in $v \in \mathbb{R}^n$, recimo u=$(a_1, a_2...a_n)$, v=$(b_1, b_2...b_n)$

VSOTA teh dveh vektorjev, ki jo zapišemo kot $u + v$, je vektor, ki ga dobimo tako, da seštejemo elemente, ki ležijo na istih indeksih vektorjev.

$$u + v = (a_1 + b_1, a_2 + b_2, ..., a_n + b_n) \text{ [2]}$$

MNOŽENJE S SKALARJEM oziroma produkt vektorja u in skalarja k, je vektor, ki ima vse elemente iz u pomnožene s skalarjem k.

$$ku = k(a_1, a_2, ...a_n) = (ka_1, ka_2, ..., ka_n) \text{ [2]}$$

Opazimo, da sta tudi $u+v$ in ku vektorja v $\mathbb{R}^n$. Vsota vektorjev različnih dimenzij ni definirana.
NEGIRANJE IN RAZLIKA sta definirana tako:

$$-u = (-1)u$$

$$u - v = u + (-v)$$

Vektor -u imenujemo nasprotni vektor u-ja in u-v imenujemo razlika u in v.
Lastnosti
Za poljubne vektorje $u, v, w \in \mathbb{R}^n$ in poljubne skalarje k,l $\in \mathbb{R}$ velja [2]:

- $(u + v) + w = u + (v + w)$
- $u + 0 = u$
- $u + (-u) = 0$
- $u + v = v + u$
- $k(u + v) = ku + kv$
- $(k + l)u = ku + lu$
- $(kl)u = k(lu)$
- $1u = u$

4.2 Matrike

Podpoglavje je povzeto po [3].
K naj bo dan kolobar. Matrika je kvadratna shema elementov iz K s tole obliko

$$A = \begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ ... & ... & ... & ... \\ a_{n1} & a_{n2} & ... & a_{nn} \end{pmatrix}$$

. In sicer je A kvadratna matrika dimenzije n x n, ker je v njej n vodoravnih vrstic in v vsaki vrstici stoji po n elementov kolobarja K. Vseh elementov a_{ij} v matriki dimenzije n x n je n^2. Indeksa i in j pri elementu a_{ij} povesta, kje leži ta element v matriki. Prvi indeks nam pove, v kateri vrstici leži, drugi pa, v katerem stolpcu. Na primer a_{23} je na križišču druge vrstice in tretjega stolpca.
Množico vseh kvadratnih matrik dimenzije n x n, sestavljenih z elementi kolobarja K, bomo zaznamovali s K_n. Zdaj bomo definirali računske operacije med matrikami, tako da bo postala množica K_n kolobar.
Seštevanje matrik
Naj bosta A in B poljubni matriki dimenzije n x n z elementi iz kolobarja K. Vsota $A + B$ je matrika dimenzije n x n, katere elementi so vsote istoležnih elementov matrike A in matrike B.
V vsoti $A + B$ stoji na preseku i-te vrstice in j-tega stolpca vsota elementov a_{ij} in b_{ij}, ki ležita na istem mestu v matrikah A in B.
Ker veljajo v kolobarju običajni zakoni za seštevanje, iz definicije vidimo, da veljata za seštevanje matrik asociativnostni in komutativnostni zakona:

$$(A + B) + C = A + (B + C)$$

in

$$A + B = B + A.$$

Matrike dimenzije n x n odštevamo tako, da odštevamo istoležne elemente. Od tod sledi, da je množica K_n vseh matrik dimenzije n x n za seštevanje Abelova

grupa. Element 0 te aditivne grupe je matrika

$$\begin{pmatrix} 0 & 0 & ... & 0 \\ 0 & 0 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & 0 \end{pmatrix}$$

, ki ima vse elemente enake 0.

Nasprotna matrika -A matrike A je matrika iz nasprotnih elementov $-a_{ij}$. Res je $A + (-A) = 0$.

Množenje matrik

Naj bosta a=$a_1, a_2...a_n$ in b=$b_1, b_2...b_n$ vektorja z n elementi poljubnega kolobarja K.

Vsoto produktov s=$a_1b_1 + a_2b_2 + ... + a_nb_n$ imenujemo **skalarni produkt** a z b. Skalarni produkt je seveda element kolobarja K.

Produkt matrike A dimenzije n x n in matrike B dimenzije n x n označimo z AB in je matrika C dimenzije n x n, pri kateri je element c_{ij}, ki leži na križišču i-te vrstice in j-tega stolpca produkta AB, enak skalarnemu produktu i-te vrstice in j-tega stolpca.

$$\begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ ... & ... & ... & ... \\ a_{n1} & a_{n2} & ... & a_{nn} \end{pmatrix} * \begin{pmatrix} b_{11} & b_{12} & ... & b_{1n} \\ b_{21} & b_{22} & ... & b_{2n} \\ ... & ... & ... & ... \\ b_{n1} & b_{n2} & ... & b_{nn} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & ... & c_{1n} \\ c_{21} & c_{22} & ... & c_{2n} \\ ... & ... & ... & ... \\ c_{n1} & c_{n2} & ... & c_{nn} \end{pmatrix}$$

pri čemer je $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + ... + a_{in}b_{nj}$

Komutativnost v splošnem ne velja, čeprav je osnovni kolobar K komutativen. Veljata pa asociativnostni in distributivnostni zakon. Torej velja

- $(AB)C = A(BC)$
- $C(A * B) = CA + CBin(A + B)C = AC + BC$

Naj ima kolobar K element 1. V tem primeru je tudi v K_n identiteta I, to je

matrika

$$I = \begin{pmatrix} 1 & 0 & ... & 0 \\ 0 & 1 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & 1 \end{pmatrix}$$

, ki ima v glavni diagonali povsod element 1 kolobarja K, na vseh drugih pa je element 0.
Velja:

$$AI = IA = A$$

za vsako matriko A dimenzije n x n.
Zamenjajmo v matriki A vodoravne vrstice s stolpci, torej zrcalimo A čez glavno diagonalo. Tako dobimo **transponirano** matriko matrike A, ki jo označimo z A^T
Torej:

$$A^T = \begin{pmatrix} a_{11} & a_{21} & ... & a_{n1} \\ a_{12} & a_{22} & ... & a_{n2} \\ ... & ... & ... & ... \\ a_{1n} & a_{2n} & ... & a_{nn} \end{pmatrix}$$

Transponiranost je vzajemna. K matriki A^T transponirana matrika je prvotna matrika A.

$$A^{TT} = A$$

Vidimo tudi, da je transponirana matrika vsote enaka vsoti transponirank sumandov:

$$(A + B)^T = A^T + B^T$$

Potence matrike

Naj bo A matrika dimenzije n x n z elementi iz kolobarja K. **Potence** matrike A so definirane tako:
$A^2 = AA$, $A^3 = A^2A$,...$A^{n+1} = A^nA$,... in $A^0 = I$

Množenje matrike z vektorjem

Produkt matrike A dimenzije n x n in vektorja x z n elementi označimo z Ax. Rezultat je vektor y, ki ima za i-ti element skalarni produkt i-te vrstice matrike z vektorjem x.

Torej: $$\begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ ... & ... & ... & ... \\ a_{n1} & a_{n2} & ... & a_{nn} \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ ... \\ y_n \end{pmatrix},$$
pri čemer je $y_i = a_{i1}x_1 + a_{i2}x_2 + ... + a_{in}x_n$.

Pri uporabi bi namesto + in * lahko uporabili kakšen drug kolobar, na primer (min,+).

Takrat bi veljalo $y_i = min\{a_{i1} + x_1, a_{i2} + x_2, ..., a_{in} + x_n\}$.

Poglavje 5

Algoritmi

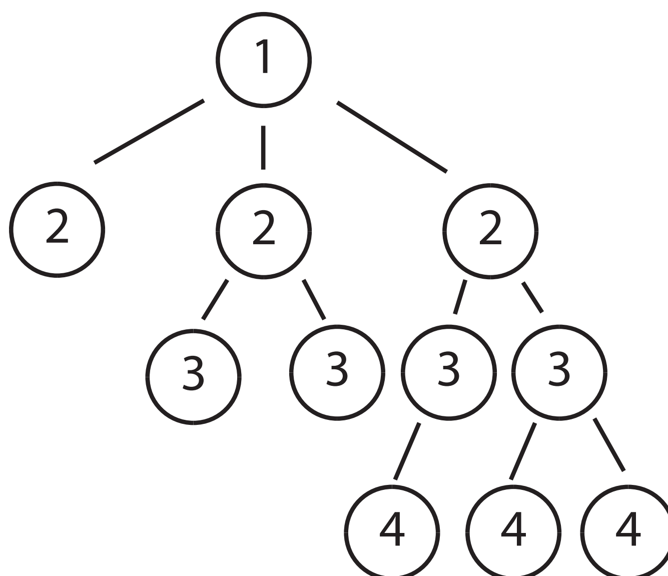
Vsi algoritmi v poglavju so implementirani po [1], prav tako so vsi opisi algoritmov in delovanja algoritmov povzeti po [1], če ni drugače zapisano.

5.1 BFS - iskanje v širino

Iskanje v širino je ena od strategij preiskovanja grafa, kjer je iskanje po grafu omejeno na dve operaciji

- obisk in preiskovanje vozlišča grafa,
- narediti en korak od trenutnega preiskovanega vozlišča do vseh najbližjih sosedov vozlišča.

Lahko bi rekli, da algoritem deluje tako, da na vsakem koraku "razširi" že obiskano območje en nivo vozlišč naprej, kjer nivo pomeni najbližje sosede [9] .
Na sliki 5.1 vidimo, kako po vrsti preiskujemo vozlišča po pravilu "najprej sosede". Pri preiskovanju grafa se lahko osredotočimo na iskanje marsičesa, v našem primeru pa nas bo zanimalo, do katerih vozlišč lahko pridemo iz začetnega vozlišča v r korakih. Če bi r bil 1, bi odgovor bil: najbližji sosedi (oddaljeni za 1 korak).
Pri reševanju tega problema je pomembna dualnost, ki obstaja med osnovno operacijo v linearni algebri (množenje vektorja in matrike) z iskanjem v širino (BFS) na grafu G, z začetkom v vozlišču s.



Slika 5.1: Slika nivojev pri preiskovanju v širino

Pomagamo si z zgoraj opisano matriko sosednosti in (ALI,IN) kolobarjem.

$$\begin{bmatrix} 0&0&1&0&0 \\ 0&0&0&1&1 \\ 1&0&0&1&0 \\ 0&1&0&0&1 \\ 1&1&0&0&0 \end{bmatrix} \;|.\&\; \begin{bmatrix} 0&0&1&0&0 \\ 0&0&0&1&1 \\ 1&0&0&1&0 \\ 0&1&0&0&1 \\ 1&1&0&0&0 \end{bmatrix} = \begin{bmatrix} 1&0&0&1&0 \\ 1&1&0&0&1 \\ 0&1&1&0&1 \\ 1&1&0&1&1 \\ 0&0&1&1&1 \end{bmatrix}$$

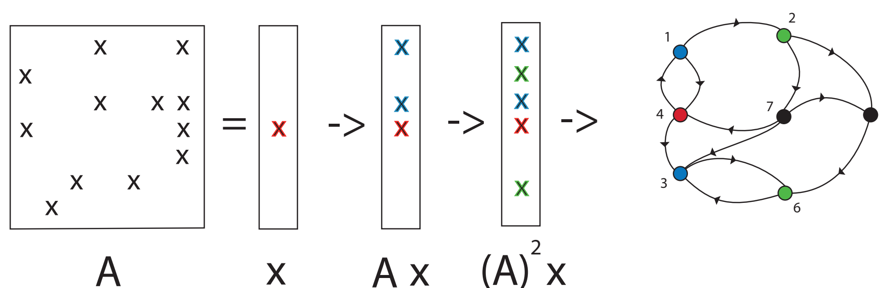
$$(0\&0)\,|\,(0\&0)\,|\,(1\&1)\,|\,(0\&0)\,|\,(0\&1) = 0\,|\,0\,|\,1\,|\,0\,|\,0 = 1$$

Slika 5.2: Primer računanja v kolobarju (ALI,IN)

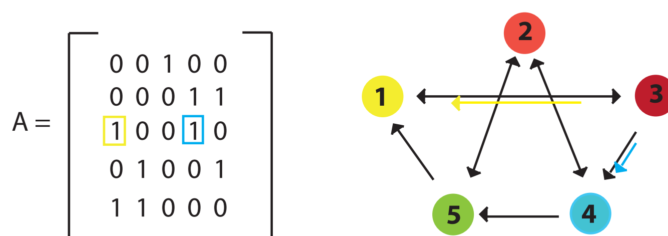
Rešitev je i-ti stolpec $G^{\diamond r}$.

Za iskanje iz vozlišča i začnemo z $x(i) = 1$, $x(j) = 0$ za $j \neq i$. Potem $y = G * x$ pobere stolpec i iz G, ki vsebuje sosede vozlišča i. Množenje med y in $G^{\diamond 2}$ da vozlišča, ki so 2 koraka stran in tako naprej, to pa je natanko iskanje v širino.

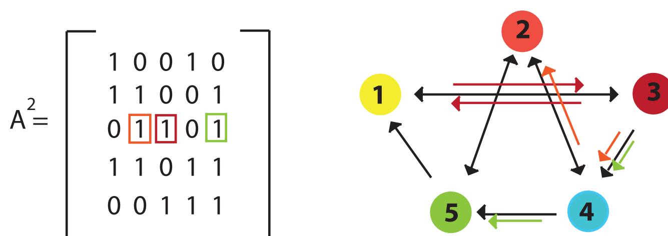
Če namesto matrike sosednosti, vzamemo matriko dosegljivosti, lahko s tem algoritmom dobimo vsa vozlišča v razdalji NAJVEČ k na k-tem koraku.



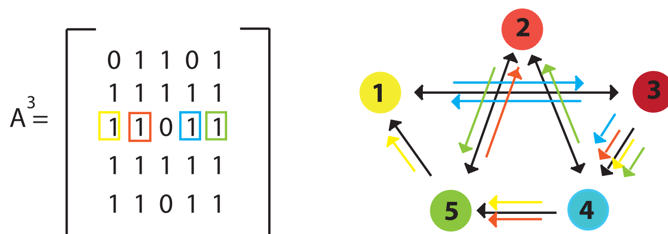
Slika 5.3: Simulacija BFS algoritma



V 1 koraku lahko iz vozlišča 3 pridemo v 1 ali 4



V 2 korakih lahko iz vozlišča 3 pridemo v 2, 3 ali 5



V 3 korakih lahko iz vozlišča 3 pridemo v 1,2, 4 ali 5

Slika 5.4: Simulacija BFS algoritma na primeru

Kot smo videli, lahko izvajamo več neodvisnih iskanj naenkrat, tako da uporabimo množenje med dvema redkima matrikama, saj se pri takem množenju izračunajo

stolpci neodvisno eden od drugega. Namesto vektorja x torej uporabimo matriko X, s stolpcem za vsako začetno vozlišče. Po množenju $Y = G * X$, stolpec j v Y vsebuje rezultat BFS za vozlišče, specificirano s stolpcem j v X. Z uporabo nepotratne podatkovne strukture za operiranje z redkimi matrikami je časovna kompleksnost iskanja enaka, kot bi bila s tradicionalno podatkovno strukturo za redek graf.

5.2 Algoritem minimalne poti

PROBLEM

Imamo usmerjen graf G z N vozlišči in M povezavami ter matriko cen C (na mestu (i j) je cena poti od vozlišča i do vozlišča j v enem koraku). Zapisi na lokacijah A(i,i) bodo 0 (cena, da pridemo do vozlišča, v katerem smo že, je 0). Želimo najti drugo matriko, ki bi imela na poziciji (i, j) zapis, ki bi povedal **najmanjšo možno ceno** pri sprehodu od i do j, pri tem pa ni pomembno, v koliko korakih bi to bilo. V bistvu iščemo najcenejše poti od nekega vozlišča s do vseh drugih vozlišč.
Problem bi že znali rešiti z iskanjem v širino, če bi imeli vse cene enake 1.

Ideja

Če je $\pi = (u_0, ..., u_{k-1}, u_k)$ najcenejša pot od u_0 do u_k, potem je $(u_0, ..., u_{k-1})$ najcenejša pot od u_0 do u_{k-1}.

Dokaz

Če bi obstajala od u_0 do u_{k-1} neka cenejša pot p, ki bi jo lahko podaljšali s korakom (u_{k-1}, u_k) in tako dobili neko pot med u_0 in u_k, ki bi bila cenejša od π, ki pa je po predpostavki najcenejša, bi zašli v protislovje [14].
Torej je vsaka najcenenjša pot podaljšek neke druge najcenejše poti. Vse, kar moramo storiti, če iščemo po grafu, je, da za vsak i najdemo njegovo predhodnico na najcenejši poti od s do i.
Porodi se ideja, da bi namesto navadnega produkta C(i,j)=$\sum_k (A(i,k) * B(k,j)$ želeli produkt v kolobarju z operacijama min in + (C=A min.+ B) $C(i,j) = min_k\{A(i,k) + B(k,j)\}$.

Zakaj nam ta operacija pride prav?

Naj bo u_i = cena najcenejše poti od začetnega vozlišča do vozlišča i

- če je $i = 1$, potem je $u_1 = 0$,
- če $i \neq 1$, potem pride najcenejša pot v i iz nekega k ($k \neq i$). Kot smo premislili zgoraj, je to del najcenejše poti do k (z ceno u_k). Ostane še vprašanje, kateri je k. K je tisti, za katerega je $u_k + c_{ki}$ minimalno. Torej: $u_i = min_{k \neq i} u_k + c_k i$, natanko to pa nam da produkt v kolobarju (min,+).

Kadar graf G nima povezave med i in j, bi po definiciji v matriki A imeli zapis A(i,j)=0. To pa bi pomenilo, da med tema vozliščema ni cene (je pot zelo poceni), mi pa bi morali nekako zagotoviti, da nam bo element v bistvu povedal, da je pot med i in j nemogoča oziroma mora biti cena taka, da se nam zagotovo nikoli ne bo splačalo iti po njej. Najlažje to storimo tako, da za take primere definiramo A(i,j)= ∞.
Poglejmo sedaj našo operacijo A min.+ A med enakima matrikama:

$$C(i,j) = min_k\{A(i,k) + A(k,j)\}$$

A(i,k) je cena na poti od vozlišča i do vozlišča j in A(k,j) je cena na poti od vozlišča k do vozlišča j. Torej je $A(i,k) + A(k,j)$ vsota cen na poti v "2 korakih", in sicer od vozlišča i do vozlišča j, ki gre skozi k, torej $min_k\{A(i,k) + A(k,j)\}$ pomeni od vseh možnih poti od i do j, ki grejo skozi različne k (gledamo po vseh možnih k) vzamemo minimalno.
Iz tega sledi, da imamo v C na poziciji i, j res najmanjšo ceno poti od vozlišča i do vozlišča j v natanko 2 korakih. Ker pa smo prej določili, da je A (i,i)=0, minimizacija za C vključuje $k = i$ in $k = j$, kar predstavlja pot od i do j v natanko 1 koraku, zato bomo raje rekli, da imamo v C(i,j) najnižjo možno ceno v NAJVEČ 2 korakih. Tudi v nadaljevanju bomo pri govoru o poti dane dolžine v bistvu govorili o tej poti.
NOTACIJA
$A^{\diamond n}$
$A^{\diamond 1} = A$
$A^{\diamond 2} = Amin. + A$
in v splošnem $A^{\diamond n} = Amin. + A^{\diamond (n-1)}$

Algoritem 5.2.10: Množenje v kolobarju (min,+): A min.+A

```
for i:=1 to visina A step 1 {
        for j:=1 to sirina A step 1 {
                minimum= Integer.MAX_VALUE;
                for k:=1 to sirina A step 1{
                        temp=A[i][k]+A[k][j];
                        if (temp<minimum) {
                                minimum=temp;
                        }
                }
                rezultat[i][j]=minimum;
        }
}
```

Kjer je:

- rezultat na začetku matrika s samimi ničlami enakih dimenzij kot A
- v minimum se shranjuje trenutni minimum za polje
- temp je trenutni zmnožek
- Integer.MAX_VALUE prikazuje ∞

$$\begin{bmatrix} 0 & 1 & 2 & \infty & \infty \\ 3 & 0 & \infty & 4 & \infty \\ \infty & 5 & 0 & \infty & 6 \\ \infty & \infty & 7 & 0 & 8 \\ 9 & \infty & \infty & 10 & 0 \end{bmatrix} \;|.\&\; \begin{bmatrix} 0 & 1 & 2 & \infty & \infty \\ 3 & 0 & \infty & 4 & \infty \\ \infty & 5 & 0 & \infty & 6 \\ \infty & \infty & 7 & 0 & 8 \\ 9 & \infty & \infty & 10 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 5 & 8 \\ 3 & 0 & 5 & 4 & 12 \\ 8 & 5 & 0 & 9 & 6 \\ 17 & 12 & 7 & 0 & 8 \\ 9 & 10 & 11 & 10 & 0 \end{bmatrix}$$

$$\min\{(3+0),(0+3),(\infty+\infty),(4+\infty),(\infty+9)\} = \min\{3,3,\infty,\infty,\infty\} = 3$$

Slika 5.5: Primer množenja v kolobarju (min,+)

A(i,j) ima **minimalno ceno od vseh poti** od i do j v 1 koraku, v bistvu obstaja samo ena tak pot, to je cena povezave med i in j. $A^{\diamond 2}$ vsebuje minimalno ceno za

največ 2 koraka in $A^{\diamond 3}$ minimalno ceno za največ 3 korake, v nadaljevanju bomo pokazali, da se ta vzorec nadaljuje tudi za višje potence. Torej obrazložimo, da vnos na mestu (i,j) v matriki $A^{\diamond n}$, ko uporabljamo kolobar (min,+), vsebuje ceno najcenejše poti od i do j z uporabo največ n korakov.

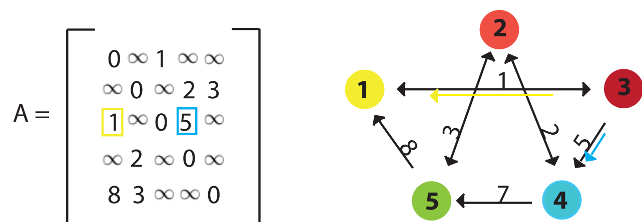
Dokaz Trditev bomo dokazali z matematično indukcijo.
Recimo, da to velja za n, pokazati pa moramo, da to velja tudi za n+1. Prav tako že vemo, da to velja za n=1, saj je A definiran tako, da je v $a_{i,j}$ cena poti med i in j (kar je pot dolžine 1)

Najcenejša pot v (največ) 1 koraku iz vozlišča i v vozlišče k je dana v A in najcenejša pot v največ n korakih iz kateregakoli vozlišča k v vozlišče j je dano v $B = A^{\diamond n}$, ampak na poti od i do j mora obstajati neko vozlišče k na najcenejši poti od i do j v največ n+1 korakih. Bellmanov princip nam pove, da mora vsaka podpot na optimalni poti tudi biti optimalna podpot. Če je torej k na optimalni poti od i do j, mora biti podpot od i do k (trivialno) optimalna in je njena cena A(i,k). Podpot v največ n korakih od k do j mora biti tudi optimalna in njena cena je B(j,k).
Izračun A min.+ B eksplicitno določa, katero vozlišče k minimizira $A(i,k)+B(k,j)$, to pa je ravno cena poti največ n+1 koraki.
MATRIČNO POTENCIRANJE
B=$A^{\diamond K}$ bi lahko izračunali z naslednjo zanko

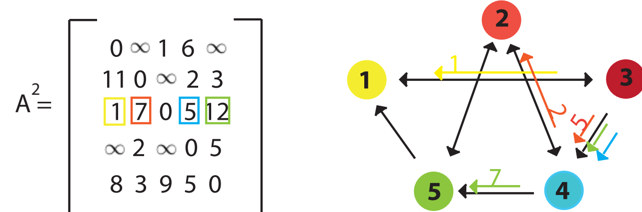
Algoritem 5.2.12: potenciranje v kolobarju (min,+)

```
B=A;
for i:=2 to K step 1{
        B=A min.+ B;
}
```

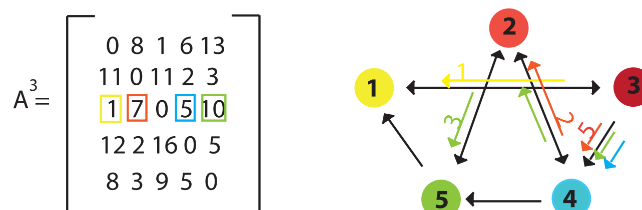
Kjer je A matrika cen.
Kot smo prej pokazali, bi B(i,j) vseboval najcenejšo pot od i do j v največ K korakih. Kaj pa, če nas zanima absolutna najcenejša pot (ne glede na število



V 1 koraku pot iz vozlišča 3 v 1 stane 1 v 4 pa 5, v ostala vozlišča ne moremo



V NAJVEČ 2 korakih lahko iz vozlišča 3 pridemo v 1 in 4 po isti ceni, 2 po 7 v 5 pa po 12



V vse ostale najkrajše poti iz vozlišča 3 ostanejo iste, razen v 5, ki sedaj stane 10

Slika 5.6: Primer iskanja minimalnih poti

korakov)? Za to ceno očitno ne bomo nikoli potrebovali več kot N-1 korakov, kjer je N število vozlišč, torej se $A^{\diamond n}$ ne spreminja več, ko n postane večji od N-1, saj pot gotovo ne bo cenejša, če bomo po istih povezavah hodili večkrat (**opomba:** vsa ta spoznanja veljajo samo za povezan graf, ki ima same pozitivne cene).

B=$A^{\diamond K}$ vsebuje samo informacijo o najmanjši ceni poti, ne pa tudi o poti sami, ta informacija se je izgubila, ko smo si v enačbi $A(i,k) + B(k,j)$ zapomnili samo minimum takih poti, nismo pa si zapomnili vozlišča k, s katerim smo dobili najcenejšo pot. Če želimo ohraniti še informacijo o poti sami, moramo pri kalkulaciji $B = A\ min. +\ B$ imeti še vzporedno matriko D, v kateri je element (i,j) vozlišče k, za katerega je bil ta minimum dosežen.

Za bolj učinkovito računanje $A^{\diamond K}$moramo najprej povedati, da za vsaka pozitivna

p in q velja

$$A^{\diamond(p+q)} = A^{\diamond p} min. + A^{\diamond q}.$$

To pa je v bistvu še ena trditev Bellmanovega principa .

Za katerokoli vozlišče k $R^{\diamond p}$ vsebuje minimalne cene poti med vozliščema i in k s p koraki (R(i,k) je cena minimalne poti med vozliščema i in k) in $S^{\diamond q}$ vsebuje minimalno ceno poti med vozliščema i in k s q koraki. (S(k,j) je cena minimalne poti med vozliščema k in j.)

Če gre minimalna pot s p+q koraki skozi neko vozlišče k, morajo biti podpoti iz vozlišča i do vozlišča k ter od vozlišča k do vozlišča j minimalne podpoti.

Torej $R\ min.+\ S$ najde vozlišče k, ki minimizira $R(i,k)+S(k,j)$. Po Bellmanovem principu mora to biti najcenejša od cen poti s p+q koraki. Račun, ki smo ga opisali, je v bistvu kar $A^{\diamond(p+q)}$.

S pomočjo te trditve lahko bolj učinkovito izračunamo $A^{\diamond K}$, in sicer, če za K izberemo najmanjšo moč dveh večjih ali enakih števil od N-1, lahko ekonomično računamo matriko, ki ima za element (i,j) absolutno minimalno ceno poti od vozlišča i do vozlišča j:

$$B^{\diamond(0)} = A$$

$$B^{\diamond(1)} = B^{\diamond(0)} min. + B^{\diamond(0)} = A^{\diamond 2}$$

$$B^{\diamond(2)} = B^{\diamond(1)} min. + B^{\diamond(1)} = A^{\diamond 4}$$

$$\ldots$$

$$B^{\diamond(r)} = B^{\diamond(r-1)} min. + B^{\diamond(r-1)} = A^{\diamond 2^r}$$

Algoritem 5.2.14: Algoritem iskanja minimalnih poti brez shranjevanja poti

```
B=A;
mnozenje=ceil(log(dolzina A)/log(2));
for i:=1 to mnozenje step 1 {
        temp=B min.+B;
        B=temp;
```

```
}
rezultat=B;
```

Kjer je:

- temp=matrika s samimi ničlami istih dimenzij kot A;
- A je matrika cen;
- min.+ je množenje v kolobarju (min,+).

Ne smemo pozabiti, da nam $A^{\diamond 2^r}$ daje cene minimalnih poti in ne poti samih, če pa želimo še poti, moramo shranjevati še serijo matrik D(m); m=1,2,...r z vrednostmi k, ki so bila vozlišča, preko katerih smo prišli do minimalne cene poti v vsakem izračunu min.+.
Lahko redefiniramo min.+ operacijo v $[C, D] = A\ min. +\ B$ tako, da hkrati računamo
$C(i,j) = min_k\{A(i,k) + A(k,j)\}$
$D(i,j) = argmin\{A(i,k) + A(k,j)\}$
ALGORITEM
$B^{\diamond(0)} = A$
$[B^{\diamond(1)}, D^{\diamond(1)}] = B^{\diamond(0)} min. + B^{\diamond(0)}$
$[B^{\diamond(2)}, D^{\diamond(2)}] = B^{\diamond(1)} min. + B^{\diamond(1)}$
.....
Če od tod želimo dobimi celotno optimalno pot iz i do j, pogledamo najprej v $D^{\diamond(r)}(i,j)$, tu najdemo k, ki je na "polovici" poti med vozliščema i in j. Potem v $D^{\diamond(r-1)}$ pogledamo v$D^{\diamond(r-1)}(i,k)$, tu najdemo vozlišče, ki je na polovici med vozliščema i in k, in v $D^{\diamond(r-1)}(k,j)$, da najdemo vozlišče na polovici med vozliščema k in j in tako dalje rekurzivno, dokler ne najdemo celotne poti (odkrivamo sosedna vozlišča po poti).

5.3 Bellman-Fordov algoritem

Bellman-Fordov algoritem rešuje problem iskanja minimalnih poti **iz enega vozlišča**. Recimo, da imamo dan graf G=(V,E), s cenami povezav w in začetnim vozliščem $s \in V$. Z Bellman-Fordovim algoritmom določimo, če imamo v grafu pot z negativnimi cenami povezav na poti, ki vsebuje s,saj v tem primeru ne moremo najti prave minimalne poti. Če v grafu nimamo negativnih povezav, algoritem vrne minimalno razdaljo poti med vozliščem s in vozliščem v za vse $v \in V$ in pripadajoče poti. Isti problem rešuje tudi na primer Dijkstrov algoritem, ki ni učinkovit za grafe, ki vsebujejo negativne povezave, saj Dijkstra požrešno izbira povezave z najnižjimi cenami in jih dodaja v drevo minimalnih poti. Bellman-Fordov algoritem vsakič pregleda vse povezave, zato je počasnejši, a uporabnejši v primeru grafov z negativnimi povezavami.
Algoritem lahko implementiramo na več različnih načinov.

5.3.1 Standardna implementacija

Za vsako vozlišče shranimo oceno minimalne razdalje d(v), kjer vzdržujemo pravilo, da $d(v) >= \Delta(s, v)$, kjer je $\Delta(s, v)$ cena poti med vozliščema s in v. Algoritem ponavlja zaporedje sprostitev vozlišč, po katerih velja $d(v) = \Delta(s, v)$. Sprostiti vozlišče pomeni, da d(v)= min{d(v),d(u)+W(u,v)}. V glavnem algoritem sestavlja N iteracij, pri katerih sprostimo vsa vozlišča v vsaki iteraciji (v poljubnem vrstnem redu). V $\pi(x)$ shranimo starša (predhodnika) tega vozlišča v drevesu minimalnih poti. π na koncu lahko uporabimo tudi za restavriranje celotne minimalne poti za neko vozlišče, ne samo za pridobitev razdalj (cene) posameznih poti.

Algoritem 5.3.16: Standardna implementacija Bellman-Fordovega algoritma

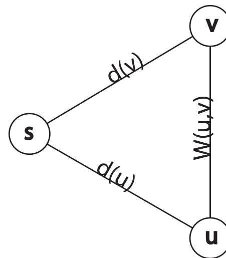
```
foreach v iz V {
        d(v)=Integer.MAX_VALUE;
        pi (v)=NIL;
}
```

```
d(s)=0;
for k:=1 to N–1 step 1 {
        foreach edge(u,v) iz E{
                Relax(u,v);
                if (d(v)>d(u) + W(u,v)){
                        d(v)=d(u) + W(u,v);
                        pi (v) =u;
                }
        }
}
foreach edge(u,v) iz E{
        if (d(v)>d(u) + W(u,v)){
                vrni "obstajajo negativne povezave"
        }
}
```

Kjer je:

- V množica vozlišč;
- E množica povezav;
- W(u,v) cena povezave med u in v;
- edge(u,v) je povezava med u in v;
- Relax(u,v) funkcija, ki sprosti vozlišče v;
- Integer.MAX_VALUE predstavlja ∞.

Na koncu preverjamo, ali negativne povezave obstajajo, in sicer, če je cena poti do v strogo večja od cene poti do u + cena povezave med u do v, potem mora vmes obstajati negativna povezava, saj v primeru, da ni negativnege povezave d(v), to ni cena minimalne poti, kot jo je označil algoritem, saj je pot skozi d(u) cenejša. Razlog za to je torej lahko samo negativna povezava na grafu.



5.3.2 Implementacija z linearno algebro

Ta implementacija nas bo bolj zanimala.
Algebrska formulacija algoritma temelji na interpretaciji z dinamičnim programiranjem.

Definicija

$$\mathbf{\Delta_k(u,v)} = \begin{cases} min\{w(p) : u \to^p v\}; & \text{če obstaja pot dolga } a(\leq k) \text{ med u in v} \\ \infty; & \text{sicer} \end{cases}$$

je minimalna cena poti med u in v z uporabo največ k vozlišč.

Če je $\Delta_k(s,v) < \Delta_{N-1}(s,v)$ za katerikoli v, potem obstaja negativni cikel, drugače velja $\Delta_k(s,v) = \Delta_{N-1}(s,v)$.
Računanje $\Delta_k(s,v)$ je logično enako sprostitvi vseh vozlišč, povezanih z v.
Natančneje: $\Delta_k(s,v) = min_u\{\Delta_{k-1}(s,u) + W(u,v)\}$. Opazimo, da lahko Δ_k izračunamo samo iz Δ_{k-1}, zato lahko vse ostale Δ_i zanemarimo.
Za predstavitev algoritma z matrično vektorskimi operacijami uporabimo (redko) sosedno matriko dimenzije NxN za shranjevanje tež povezav (matrika cen) in 1xN vektor za shranjevanje minimalnih razdalj poti v a $(\leq k)$ korakih $\Delta_k(s,*)$. Na diagonali matrike cen imamo ničle (če smo že v točki i, nas ne stane nič, da pridemo v točko i).
Formulo za $\Delta_k(s,v)$ lahko prevedemo v matrični produkt precej direktno.
Imamo $d_k(v) = min_{u \in N}(d_{k-1}(u) + A(u,v))$, kar je ravno produkt v polkolobarju (min,+).

$$d_k(v) = d_{k-1} min. + A(:,v) \to d_k(v) = d_{k-1}(v) min. + A.$$

Ceno minimalne poti lahko predstavimo tudi $d = d_0 A^N$, kjer je d_0 vektor, ki ima na mestu s 0, drugje pa je ∞. Ta izraz nam prinese dva algoritma za izračun dolžine minimalne poti iz enega vozlišča.

PRVI je algebrska predstavitev Bellman-Fordovega algoritma spodaj.

Algoritem uporabi vektor velikosti reda N in množenje redkih matrik, tako da opravi O(NM) primerjav.

Algoritem 5.3.18: Bellman-Ford (A,s)

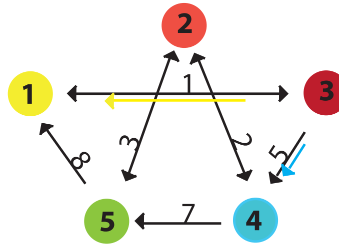
```
d=Integer.MAX_VALUE;
d(s)=0
for k:=1 to N-1 step 1 {
        d=d min.+A;
}
if (d ni d min.+A){
        vrni "obstajajo negativne povezave"
}
```

Kjer je:

- d vektor, ki pove, za katero vozlišče s nas zanimajo minimalne poti;
- d na koncu rešitev, če ne obstaja negativni cikel;
- A matrika cen.

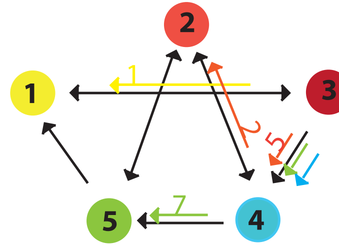
DRUGI deluje tako, da najprej naračunamo $A^{\diamond N}$ z zaporednim kvadriranjem. Opazimo, da ta algoritem v bistvu izračuna najprej vse možne kombinacije minimalnih poti med vozlišči, potem pa produkt d min.+ A samo izbere vrstico iz matrike $A^{\diamond N}$. Algoritem zahteva O(N^3 logN) dela. Algebrski (prvi) algoritem Bellman-Forda ima asimptotično boljšo zmogljivost od zaporednega kvadriranja v najslabšem primeru, a je pri zaporednem kvadriranju več možnosti paralelizma kot pri prvem algoritmu.

$$d' = d \text{ min.+} A = \begin{bmatrix} 1 \\ \infty \\ 0 \\ 5 \\ \infty \end{bmatrix}^T$$



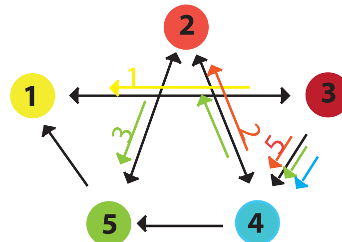
V 1 koraku pot iz vozlišča 3 v 1 stane 1 v 4 pa 5, v ostala vozlišča ne moremo

$$d'' = d' \text{ min.+} A = \begin{bmatrix} 1 \\ 7 \\ 0 \\ 5 \\ 12 \end{bmatrix}^T$$



V NAJVEČ 2 korakih lahko iz vozlišča 3 pridemo v 1 in 4 po isti ceni, 2 po 7 v 5 pa po 12

$$d''' = d'' \text{ min.+} A = \begin{bmatrix} 1 \\ 7 \\ 0 \\ 5 \\ 10 \end{bmatrix}^T$$



V vse ostale najkrajše poti iz vozlišča 3 ostanejo iste, razen v 5, ki sedaj stane 10

Slika 5.7: Primer izračuna po Bellman-Fordovem algoritmu

Algoritem 5.3.20: Bellman-Ford (A,s)

```
d=Integer.MAX_VALUE;
d(s)=0;
B=A;
mnozenje=ceil(log(dolzina A)/log(2));
for i:=1 to mnozenje step 1{
        temp=B min.+B;
        B=temp;
}
```

```
d=d min.+B;
if (d ni d min.+B){
        vrni "obstaja negativna povezava"
}
```

Opazimo, da je iskanje minimalne poti po Bellman-Fordu podobno iskanju te poti po algoritmu minimalne poti, le da tu iščemo minimalno pot in ceno te poti samo za vozlišče, določeno z vektorjem d. Pri prvem načinu res računamo samo en vektor, pri drugem načinu je izračun matrike isti kot pri algoritmu minimalne poti, zadnji korak pa je v bistvu prvi korak prvega načina (izberemo vrstico, ki prikazuje cene poti za želeno vozlišče).

5.3.3 Računanje drevesa minimalnih poti

Kot dodatek k računanju cen minimalnih poti Bellman- Fordov algoritem najde tudi minimalno pot. Te poti tipično shranjujemo s pomočjo π, preko katerega potem lahko izračunamo drevo minimalnih poti. Algoritem lahko torej tudi razširimo, tako da nam že sam vrne drevo minimalnih poti.
Opis in logika iskanja drevesa minimalnih poti, ki bo opisana v nadaljevanju, se lahko trivialno aplicira tudi na algoritem Floyd-Warshall.
Če v grafu nimamo ciklov dolžine 0, je računanje drevesa minimalnih poti precej enostavno.
Nastavimo $\pi(v) = u$, tako da $d(u) + W(u, v) = d(v)$ za vsak $v \neq s$ in $\Delta(s, v) \neq \infty$. Ker $d(v) = d(u) + W(u, v)$ za nekatere $u \neq v \in V$ in $d(v) \geq (d(u) + W(u, v))$ za vse $u \in V$, je $\pi(v) = argmin_{u \neq v}\{d(u) + W(u, v)\}$.
Ta izraz lahko algebrsko predstavimo kot:

$$\pi = d\ argmin. + \ (A + diag(\infty)),$$

kjer $diag(\infty)$ pomeni, da ima matrika po diagonali vrednosti ∞, s tem izključimo zanke v grafu.
Opazimo, da $\pi(s)$ nima vrednosti, zato nastavimo $\pi(s) = NIL$.

Podobno za vsak v z $\Delta(s,v) = \infty$, vrednost $\pi(s)$ ni pravilna, kar moramo tudi odpraviti. Če privzamemo, da je vsako vozlišče na grafu dosegljivo iz s, potem ne obstaja vozlišče, da bi veljalo $\Delta(s,v) = \infty$.

V grafu z zankami uporaba enačbe $\pi = d\ argmin. + \ (A + diag(\infty))$ najbrž ne bo pripomogla k iskanju drevesa minimalnih poti, ker lahko argmin izbere vozlišča, ki formirajo zanko. Zato želimo delati na grafu brez zank.

Za izračun kazalcev na starše na grafih brez zank imamo 2 pristopa.

PRVI pristop prilagodi naše operatorje za računanje na terkah s tremi elementi, namesto na utežeh realnih vrednosti. Ta pristop sledi standardnemu Bellman-Fordovemu algoritmu, tako da posodablja kazalce na starše z večanjem razdalje med vozlišči.

Prednosti: ves čas se uporablja samo en polkolobar, v katerem računamo za vse operacije, tako da je v izračune treba vpeljati le terke in na podlagi tega spremeniti operacije iz algebrskega Bellman-Fordovega algoritma.

DRUGI pristop uporablja idejo iz enačbe $\pi = d\ argmin. + \ (A + diag(\infty))$, in sicer tako, da sledi izračunu drevesa minimalnih poti do konca.

Prednosti: uporablja enostavnejše terke z dvema elementoma, ampak je treba računati v različnih polkolobarjih. Pristop vključuje dodatne popravke za odpravo nepravilnih elementov v π.

5.3.4 Računanje kazalcev na starše vozlišč

Računanje kazalcev na starše vozlišč je glavna razlika med osnovnim Bellman-Fordovim algoritmom in računanjem drevesa minimalnih poti.

V originalnem Bellman-Fordovem algoritmu so kazalci na starše vozlišč posodobljeni le, ko sprostimo povezavo in je nova razdalja strogo krajša, kot že prej znana.

Naj bo Π_k ena od možnih vrednosti, ki jo lahko dodelimo kazalcu na starša vozlišča v k-ti iteraciji Bellman-Fordovega algoritma.

Potem je

$$\Pi_k(s,v) = \begin{cases} NIL; & k = 0 \ \& \ v = s \\ 0; & \text{k=0} \ \& \ v \neq s \\ \Pi_{k-1}(s,v); & k \geq 1 \ \& \ \Delta_k(s,v) = \Delta_{k-1}(s,v) \\ \{u : \Delta_k(s,v) = \Delta_{k-1}(s,v) + W(u,v)\}; & sicer \end{cases}$$

Po premisleku nam to pravilo da naslednjo lemo, ki pove, da je starš, ki ga algoritem izbere, res starš na minimalni poti, v največ k korakih, ki je najcenejša.

Lema 5.3.1 *Recimo, da imamo graf z določenim vozliščem, ki je začetek poti in brez zank.*
Potem $u \neq NIL \in \Pi_k(s,v)$, *če in samo če obstaja a in pot p =(s,...u,v) taka, da* $w(p) = \Delta_k(s,v)$ *in ne obstaja* $s \to^{p'} v$, *da bi veljalo* $w(p') = \Delta_k(s,v)$ *in* $\mid p' \mid < \mid p \mid$.
Nadalje $\Pi_k(s,s) = NIL$.

Dokaz Lemo bomo dokazali z indukcijo.
Lema trivialno drži pri k=0.
Če $\Delta_k(s,v) = \Delta_{k-1}(s,v)$, potem minimalna pot v največ k korakih iz vozlišča s v vozlišče v stane največ toliko, kolikor je v k-1 korakih in lema velja pri $\Pi_k(s,v) = \Pi_{k-1}(s,v)$.
Če $\Delta_k(s,v) < \Delta_{k-1}(s,v)$, potem vsebujejo vse poti v največ k korakih iz vozlišča s v vozlišče v natanko k korakov in lema velja po definiciji Π_k .
Iz $\Delta_k(s,s) = 0$ sledi da $\Pi_k(s,s) = NIL$.

Lema 5.3.1 je uporabna za oba pristopa izračuna drevesa minimalnih poti.

RAČUNANJE KAZALCEV NA STARŠE S 3-TERKAMI

Glede na to, da je originalen Bellman-Fordov algoritem pravilen, če sproščamo povezave v poljubnem vrstnem redu, lema pove, da lahko posodabljamo kazalec na starša vozlišča tako, da je kazalec predzadnje vozlišče na **katerikoli minimalni poti**, saj so vse zadnje povezave na enako dolgi poti sproščene v isti iteraciji. Torej, če upoštevamo dolžino poti in ceno poti, ko opravljamo posodobitev kazalcev, ne potrebujemo stroge neenakosti v enačbi za izračun kazalca starša.

Cilj je, da povežemo razdaljo in kazalce na starše vozišč s spreminjanjem naših operacij tako, da delujejo na 3-terkah. Te 3-terke so sestavljene iz naslednjih komponent: celotna cena poti, dolžina poti in predzadnje vozlišče. S pametno posodobitvijo operatorjev lahko še vedno uporabimo iste algoritme, ki temeljijo na že prej obrazloženi algebri.

Definirajmo naše skalarje kot 3-terke v obliki $(w, h, \pi) \in S = (\mathbb{R}_\infty \times N \times V) \cup \{(\infty, \infty, \infty), (0, 0, NIL)\}$, kjer $\mathbb{R}_\infty = \mathbb{R} \cup \{\infty\}$ in $\mathbb{N}$ =1,2,3,.... Vrednost (∞, ∞, ∞) pomeni, da pot ne obstaja, in vrednost (0,0,NIL) pomeni pot od vozlišča do samega sebe. Prva vrednost $w \in \mathbb{R}_\infty$ je cena poti, $h \in \mathbb{N}$ je dolžina poti oziroma število iteracij, $\pi \in V$ je predzadnje vozlišče na poti.

Glede na definicijo teh 3 vrednosti je polnjenje sosedne matrike naslednje:

$$A(u, v) = \begin{cases} (0, 0, NIL); & u = v \\ (W(u, v), 1, u); & u \neq v \ \& \ (u, v) \in E \\ (\infty, \infty, \infty); & (u, v) \notin E \end{cases}$$

Opazimo, da je polje v matriki v bistvu določeno s ceno povezave. Tudi osnovni algoritem že shranjuje u v neki strukturi, torej pri implementiranju te različice algoritma ni nujno, da se nam poveča količina prostora, ki ga potrebujemo za sosedno matriko.

Za nastavljanje osnovnega vektorja razdalje d_0, ki nam pove, za katero vozlišče nas zanimajo minimalne poti, velja:

$$d_0(v) = \begin{cases} (0, 0, NIL); & v = s \\ (\infty, \infty, \infty); & sicer \end{cases}$$

Za dodatno operacijo, o kateri smo že govorili, uporabimo operator lmin, ki je definiran kot leksikografski minimum in primerja prvo vrednost 3-terke takole:

$$lmin(w_1, h_1, \pi_1), (w_2, h_2, \pi_2)) = \begin{cases} (w_1, h_1, \pi_1); & w_1 < w_2 \vee (w_1 = w_2 \wedge h_1 < h_2) \vee \\ & (w_1 = w_2 \wedge h_1 = h_2 \wedge u_1 < u_2) \\ (w_2, h_2, \pi_2); & sicer \end{cases}$$

Brez izgube za splošnost lahko predpostavimo, da so vozlišča števila 1,2,...N in NIL manjše od ∞ za vse $v \in V$.

Za vse operacije z množenjem definiramo novo binarno funkcijo $+_{rhs}$. Ta funkcija doda prvi dve vrednosti terke in ohrani tretjo vrednost terke argumenta, ki je na desni strani operanda.

$$(w_1, h_1, \pi_1) +_{rhs} (w_2, h_2, \pi_2) = \begin{cases} (w_1 + w_2, h_1 + h_2, \pi_2); & \pi_1 \neq \infty \ \& \ \pi_2 \neq NIL \\ (w_1 + w_2, h_1 + h_2, \pi_1); & sicer \end{cases}$$

Izjemo za $\pi_1 = \infty$ potrebujemo, da imamo identiteto za množenje, kadar je ta operator uporabljen pri množenju v konjunkciji v lmin za seštevanje. Izjema za $\pi_2 = NIL$ je popravek.
Opazimo, da v nasprotju z običajnim +, $+_{rhs}$ ni komutativna.

Lema 5.3.2 *S=$(\mathbb{R}_\infty \times N \times V) \wedge \{(\infty, \infty, \infty), (0, 0, NIL)\}$ nad operatorjem lmin.$+_{rhs}$ je polkolobar.*

Pravilnost algoritma, ki bazira na terkah, dokazuje naslednja lema. Spomnimo se, da je bil cilj transformacije na 3-terke ohraniti zvezo $d_n = dA^{\diamond N}$, kjer sta seštevanje in množenje definirana z lmin in $+_{rhs}$.

Lema 5.3.3 *Recimo, da imamo graf z določenim vozliščem s. Naj bosta matrika sosednosti A in začetni vektor d_0 določena po zgornjih enačbah za A in d_0 in naj bo $d_k = d_{k-1} lmin. +_{rhs} A$; $k \geq 1$. Če je $h \leq k \in \mathbb{N}$ minimalno število iteracij (dolžina poti) minimalne poti (v največ k korakih) iz vozlišča s v vozlišče v, potem $\exists u \in V$ tako da $d_k(v) = (\Delta_k(s, v), h, u) \wedge u = min\{v : v \in \Pi_k(s, v)\}$ in obstaja pot dolžine h; p=(s,...u,v) z $w(p) = \Delta_k(s, v)$.*
Če ne obstaja pot z največ k koraki, potem $d_k(v) = (\infty, \infty, \infty)$. Za začetno vozlišče velja $d_k(s) = (0, 0, NIL)$.

RAČUNANJE KAZALCEV NA STARŠE NA KONCU

Pri tem pristopu sledimo enaki ideji; spremenimo le skalarje, tako da delujejo na zankah. Ta pristop še vedno uporablja terke, ampak enostavnejše.
Uporabimo 2-terke $(w, h) \in \mathbb{N}_\infty \times (\mathbb{N} \cup \{0, \infty\})$. Vrednosti imajo isti pomen kot pri 3-terkah, samo da tu v terki nimamo kazalca na starša. Nastavljanje A in d_0 je očitno.
Bellman-Fordov algoritem za 2-terke ostaja enak kot za 3-terke, le da zdaj uporabimo funkcijo lmin namesto navadnega seštevanja in enostavnejšo + namesto množenja. Z drugimi besedami: $d_k = d_{k-1}\ lmin. +\ A$. Pravilnost sledi iz leme 5.3.3 kot pri prvem načinu, ker so operatorji povsem enaki kot pri 3-terkah, le da tu ignoriramo tretji element.

Za izračun kazalcev na starše vozlišč nastavimo: $\pi = d_n\ argmin.+\ (A{+}diag((\infty,\infty)))$. Naslednja lema trdi, da so vse vrednosti v $\pi(v)$ pravilne za vsa vozlišča $v \neq s$, ki so dosegljiva iz vozlišča s.

Lema 5.3.4 *Recimo, da imamo graf z določenim začetnim vozliščem s in brez zank. Naj bo* $d_n(v) = (\Delta_k(s,v), h_v)$*, kjer je* h_v *najmanjša cena najcenejše poti iz vozlišča s v vozlišče v. Naj bo* $\pi = d_n\ argmin. + (A + diag((\infty,\infty)))$*. Potem za vse* $v \in V{-}s$*, za katere velja* $\Delta_k(s,v) < \infty$*, imamo* $\pi(v)$ *in* $\Pi_n(s,v)$*.*

Dokaz Recimo, da so vsa vozlišča $v \in V{-}s$ dosegljiva iz vozlišča s. Produkt $\pi = d_n\ argmin.{+}(A{+}diag((\infty,\infty)))$ da $\pi(v) = argmin_{u \notin v}\{(\Delta(s,u){+}W(u,v), h_u{+}1)\}$. Po lemi 5.3.1 imamo $u \in \Pi_n(s,v)$, če in samo če $\Delta(s,u) + W(u,v) = \Delta(s,v)$, in $h_u = h_v - 1$.
Torej $\pi(v) = argmin_{u \in \pi_n(s,v)}\{(\Delta(s,v), h_v)\}$, ki nam da rezultat $\pi(v) = \Pi_n(s,v)$.

Za vsa vozlišča, ki niso dosegljiva iz vozlišča s, ta pristop ne vrača pravega rezultata, prav tako ne za vozlišče s samo, tako da moramo uvesti dodatne popravke, da dosežemo, da π vsebuje pravilne vrednosti. Alternativno lahko spremenimo argmin, da vzame vrednost ∞, če so vsi operandi (∞,∞) in potem je edini popravek, ki ga moramo upoštevati, da nastavimo $\pi(s) = NIL$.

5.4 Floyd-Warshallow algoritem

Algoritem rešuje problem iskanja najcenejših poti za vse možne dvojice vozlišč. Recimo, da imamo graf G = (V,E) z utežmi na povezavah w in brez zank, potem ta algoritem vrne cene minimalnih poti $\Delta(u,v)$ za vse $u, v \in V$. Tako kot Bellman-Fordov je Floyd-Warshallow algoritem rešitev z dinamičnim programiranjem. Brez izgube za splošnost lahko označimo vozlišča z 1,2,...N.
Algoritem uporabi $D_k(u,v)$ za predstavitev minimalne poti iz vozlišča u do vozlišča v z uporabo samo vmesnih vozlišč $v \in 1, 2, ...N \subseteq V$.
Torej

$$D_k(u,v) = \begin{cases} W(u,v); & k = 0 \\ min\{D_{k-1}(u,v), D_{k-1}(u,k) + D_{k-1}(k,v)\}; & k \geq 1 \end{cases}$$

Čas za izvajanje je $O(N^3)$ zaradi trojne for zanke. V algoritmu, katerega psevdokoda je zapisana spodaj, shranjujemo vse D_k, čeprav bi lahko shranjevali samo D_k in D_{k-1} v vsakem koraku, torej je poraba prostora $O(N^2)$

5.4.1 Standardna implementacija Floyd-Warshallovega algoritma

Algoritem 5.4.22: Standardni Floyd-Warshallov algoritem

```
for u:=1 to N step 1{
    for v:=1 to N step 1{
        D_0(u,v)=W(u,v);
        if (u==v){
                Pi_0(u,v)=NIL;
        }
        if (u != v AND W(u,v)<Integer.MAX_VALUE){
                Pi_0(u,v)=u;
        }else {
                Pi_0(u,v)=nedefiniran;
        }
    }
}
for k:=1 to N step 1{
    for u:=1 to N step 1{
        for v:=1 to N step 1{
            if (D_(k-1)(u,k)+D_(k-1)(k,v) <= D_(k-1)(u,v)){
                D_k(u,v)=D_(k-1)(u,k)+D_(k-1)(k,v);
                Pi_k(u,v)=Pi_(k-1)(k,v);
            }else{
                D_k(u,v)=D_(k-1)(u,v);
                Pi_k(u,v)=Pi_(k-1)(u,v);
            }
```

```
            }
        }
}
```

Kjer je

- Pi_k je predzadnje vozlišče na najcenejši poti, kjer lahko uporabimo le vozlišča od 1 do k.

Element Pi_k ni enak tistemu pri Bellman-Fordovem algoritmu, kjer je element množica predzadnjih vozlišč na vsaki minimalni poti, ki ima dolžino največ k od vozlišča u do vozlišča v.
Če definiramo $\pi_k(v) = \Pi_k(s, v)$, potem π_k vključuje drevo minimalnih poti iz vozlišča s. Celotna množica, zajeta v π_k, pa ne formira nujno drevesa.

5.4.2 Algebrska implementacija Floyd-Warshallovega algoritma

Pri implementaciji algoritma z matričnimi in vektorskimi operacijami uporabimo matriko D_n dimenzije $N \times N$ za shranjevanje cen poti (matriko cen). Inicializiramo $D_0 = A$ oziroma $D_0(u, v) = W(u, v);\ D_0(u, u) = 0$
Rekurzivna definicija D_0 upošteva 2 možnosti, in sicer, ali minimalna pot iz vozlišča u do vozlišča v, ki lahko vsebuje vozlišča od 1 do k, **vsebuje vozlišče k ali ne**.
Algoritem deluje tako, da izračuna teže minimalnih poti, ki vsebujejo vozlišče k za vse pare vozlišč (u,v) naenkrat. Natančneje: k-ti stolpec D_k oziroma $D_k(k, :)$ vsebuje cene minimalnih poti, ki gredo skozi vozlišče k. Podobno k-ta vrstica $D_k(k, :)$ vsebuje cene minimalnih poti iz vozlišča k v katerokoli vozlišče u. Torej je vse, kar moramo narediti, da dodamo vsak par teh vrednosti, tako dobimo cene minimalnih poti, ki gredo skozi vozlišče k, in to storimo tako, da računamo:

$$D_k = D_{k-1} min. + (D_{k-1}(:, k) min. + D_{k-1}(k, :))$$

Algoritem izvede N množenj vektorjev, zato je njegova kompleksnost $O(N^3)$.

Algoritem 5.4.24: Algebrski Floyd-Warshallov algoritem

```
D=A;
for k:=1 to N step 1{
        D=D min.+ (D(:,k) min.+D(k,:));
}
```

Kjer je :

- A matrika cen;
- min.+ je množenje v polkolobarju (min,+), opisan pri Bellman-Fordovem algoritmu;
- D(:,k) matrika, ki vsebuje vse vrstice in k-ti stolpec;
- D(k,:) matrika, ki vsebuje vse stolpce in k-to vrstico.

5.4.3 Rekonstrukcija minimalne poti

Poenostavljeno minimalno pot rekonstruiramo tako, da si vzporedno z matriko cen D shranjujemo matriko PATH, ki je z D povezana tako:

- D(i,j)=cena poti med i in j;
- PATH(i,j)=predhodno vozlišče k vozlišča j, preko katerega smo prišli iz vozlišča i do vozlišča j po minimalni poti;
- če PATH(i,j)=0 in D(i,j)= ∞ to pomeni, da od vozlišča i do vozlišča j ne obstaja nobena pot;
- če PATH(i,j)=0 in D(i,j) $\neq \infty$ to pomeni, da na minimalni poti med vozliščema i in j ni vmesnega vozlišča, minimalna pot je kar povezava med vozliščema.

Vzporedno vodenje matrike PATH implementiramo tako, da se vsakič, ko se spremeni minimalna pot v polju D(i,j) spremeni tudi polje PATH(i,j), in sicer vstavimo vozlišče, skozi katerega ravnokar ugotavljamo minimalno pot.

Algoritem 5.4.26: Vzdrževanje vozlišč na minimalni poti

```
for m:=1 to N-1 step 1{
    for k:=1 to N step 1 {
        for i:=1 to N step 1 {
            for j:=1 to N step 1{
                if(A[i][k]+A[k][j]<A[i][j] ){
                        A[i][j]=A[i][k]+A[k][j];
                        path[i][j]=k;
                }
            }
        }
    }
}
```

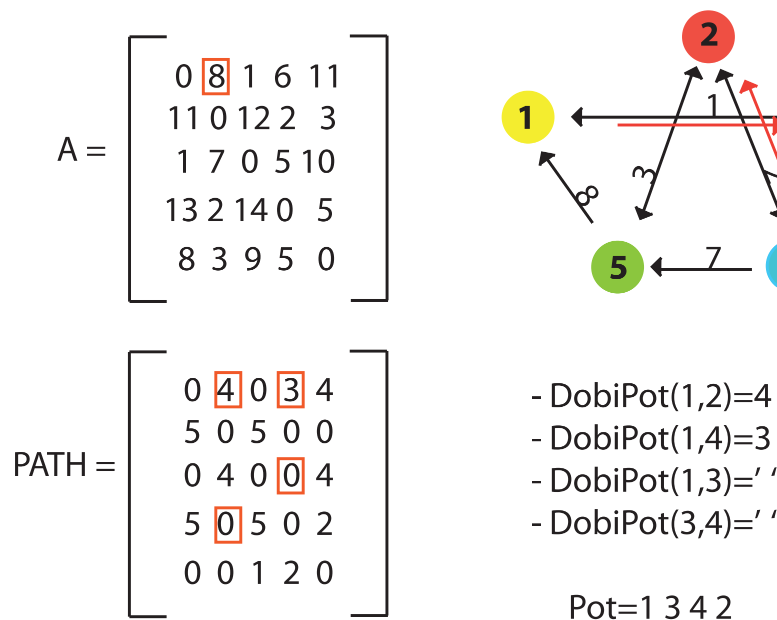
Kjer je:

- A matrika cen;
- path matrika predhodnih vozlišč na minimalni poti.

Algoritem 5.4.28: Rekonstrukcija minimalne poti med i in j (DobiPot(i,j)

```
vmesnoVozlisce=path[i][j];
if (vmesnoVozlisce =0 & A[i][j] = Integer.MAX_VALUE){
        Ni-poti;
}
if (vmesnoVozlisce =0 & A[i][j] ni Integer.MAX_VALUE){
        vrni-praznoPolje;
```

```
}else{
        vrni− DobiPot(i,vmesnoVozlisce)+vmesnoVozlisce
        +DobiPot(vmesnoVozlisce,j);
}
```



Slika 5.8: Primer rekonstrukcije poti po Floyd-Warshallovem algoritmu

5.5 Primov algoritem

5.5.1 Minimalno vpeto drevo

Definicija *DREVO* je v teoriji grafov graf, v katerem sta poljubni dve točki povezani z natanko eno enostavno potjo. Drevo je v bistvu vsak povezan graf brez ciklov. **Gozd** je nepovezana unija dreves [9].

Definicija *VPETO DREVO* T povezanega neusmerjenega grafa G je drevo, ki ga sestavljajo vse točke in nekatere (ali pa morda vse) povezave grafa G. Vpeto

drevo je izbira povezav, ki tvorijo drevo preko vseh točk. Zato velja, da je število povezav vpetega drevesa enako V-1, kjer je V število oglišč grafa G. To pomeni, da vsaka točka leži na drevesu, pri tem pa ne nastane noben cikel. Vpeto drevo povezanega grafa G lahko definiramo tudi kot maksimalno množico povezav G, ki ne vsebuje ciklov, ali pa kot minimalno množico vseh povezav, ki povezuje vse točke. [9]

Teža vpetega drevesa je vsota tež (cen) povezav, ki jih drevo vsebuje. Definiramo jo takole:

$$w(T) = \sum_{e \in T} w(e)$$

Minimalno vpeto drevo je vpeto drevo, ki ima med vsemi vpetimi drevesi grafa najmanjšo težo (ceno). Veljati mora $w(T) \leq w(T')$ za vsako vpeto drevo T'.

Pri problemu minimalnega vpetega drevesa imamo dan neusmerjen graf G(E,V) s cenam povezav w: $E \rightarrow \mathbb{R}_{\infty}$, kjer $W(u,v) = \infty$, če med vozliščema u in v ni povezave. Zaradi poenostavitve je W(v,v)=0 za vse $v \in V$.

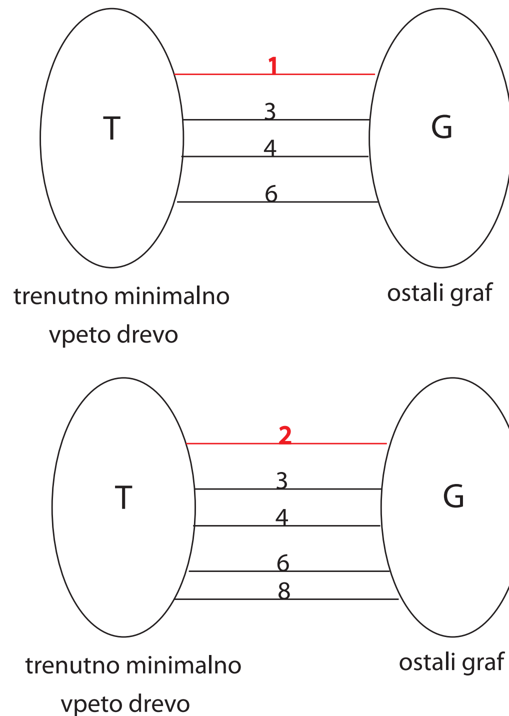
Podoben problem kot je iskanje minimalnega vpetega drevesa, je iskanje minimalnega vpetega gozda v nepovezanem grafu, ki ga lahko rešimo direktno s pomočjo problema iskanja minimalnega vpetega drevesa v vsaki nepovezani komponenti grafa. Za poenostavitev problema bomo predpostavljali, da je graf povezan.

Primov algoritem je eden od algoritmov, ki rešujejo problem minimalnega vpetega drevesa. Algebrska različica v splošnem nima tako dobrih zmogljivosti kot standardna različica.

5.5.2 Standardna implementacija algoritma

Primov algoritem rešuje problem minimalnega vpetega drevesa z ustvarjanjem množice S, ki vsebuje povezave, vsebovane v vpetem drevesu. V vsaki iteraciji v S dodamo "najbližjo" povezavo, ki še ni v S. Rečemo, da je povezava (u,v) "najcenejša" povezava, ki ni v S, če $u \in S \wedge v \notin S \wedge W(u,v) = min\{(u',v') : u' \in S \wedge v' \notin S\}$. Recimo, da je povezava (u,v) najcenejša povezava, ki je še nimamo v S in $u \in S$. Potem Primov algoritem posodobi S = S U {v} in T=T U {(u,v)}.

To ponovimo N-1 krat, na koncu je T naše vpeto drevo, S pa množica vozlišč, ki so vsebovana v vpetem drevesu.



Slika 5.9: Skica delovanja Primovega algoritma

Primov algoritem je v osnovi implementiran z uporabo prioritetne vrste. Prioritetna vrsta je podatkovna struktura, ki dinamično vzdržuje zaporedje množice objektov s ključi in podpira naslednje operacije:

- INSERT(Q,x) : vstavi objekt s ključem x v vrsto (Q=Q U {x})
- Q=BUILD-QUEUE($x_1, x_2, ..., x_n$): naredi vstavljanje objektov $x_1, x_2, ..., x_n$ v prazno vrsto Q (Q={$x_1, x_2, ..., x_n$}).
- EXTRACT-MIN(Q): izbriše in vrne element v Q z najmanjšim ključem. Če predpostavimo, da imamo enolične ključe: če ključ(x) $= min_{y \in Q}$, potem Q=Q-{x}.

- DECREASE-KEY(Q,x,k): zmanjša vrednost ključa elementa x na vrednost k, pri predpostavki, da $k \leq key(x)$. Tukaj je x kazalec na objekt, kateremu manjšamo ključ.

Naivna implementacija prioritetne vrste je neurejen seznam. Tukaj sta INSERT in DECREASE-KEY trivialna in potrebujeta čas O(1). EXTRACT-MIN je O(V) (kjer je V število vozlišč v grafu), saj moramo v najslabšem primeru preiskati celoten seznam vozlišč. Pri uporabi Fibonaccijeve kopice ima časovno kompleksnost O(M + NlogN) (kjer je M število povezav in N število vozlišč na grafu).

Algoritem 5.5.30: Standardna implementacija Primovega algoritma

```
foreach v iz V {
        key(v)=Integer.MAX_VALUE;
        pi(v)=NIL;
}

cena=0;
s=Random(s iz V);
key(s)=0;
Q=BUILD-QUEUE(V);
while (Q ni 0){
        u=EXTRACT-MIN(Q);
        cena=cena + W(pi(u),u);
        foreach u ; (u,v) iz E {
                if ( v iz Q & W(u,v) < key(v)){
                        DECREASE-KEY(Q,v,W(u,v));
                        pi(v)=u;
                }
        }
}
```

Kjer je:

- random funkcija, ki naključno izbere vozlišče s iz V;
- pi shranjuje starše vozlišča.

5.5.3 Algebrska implementacija Primovega algoritma

Uporabimo redko matriko dimenzije $N \times N$ A za shranjevanje cen povezav, vektor s dimenzije $1 \times N$ za navajanje vsebnosti v množici S in vektor d dimenzije $1 \times N$ za shranjevanje cen povezav, ki niso v S.
s in d vzdržujemo takole:

$$s(v) = \begin{cases} \infty; & v \in S \\ 0; & sicer \end{cases}$$

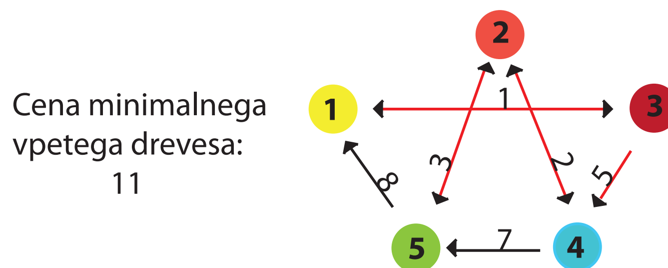
d(v)=$min_{u \in S} W(u, v)$.
Če $v \notin S$, potem d(v) poda najcenejšo povezavo, ki povezuje vozlišče v z S. Če $v \in S$, potem $d(v) = 0$. Algoritem najprej najde vozlišče, ki je najbližje S (kar pomeni, da je povezava vmes najcenejša). Ta korak izvede argmin čez celoten vektor, zato potrebuje O(N) časa za vsako iteracijo. Seštevanje vektorja zmanjša ključe vseh sosedov vozlišča u. Ta korak lahko traja O(N) časa za vsako iteracijo, ampak v skupnem vsako povezavo posodobimo le enkrat, zato je iteracij v bistvu O(M). Skupno ima torej algoritem kompleksnost $O(N^2)$.

Algoritem 5.5.32: Algebrska implementacija Primovega algoritma

```
s=0;
cena=0;
s=Random(s iz V);
s(1)=Integer.MAX_VALUE;
d=A(1,:);
while s ni Integer.MAX_VALUE{
        u=argmin{s+v};
        s(u)=Integer.MAX_VALUE;
        cena=cena+d(u);
        d=min{d,A(u,:)};
}
```

Kjer je:

- A matrika cen;
- d in s vektorja, opisana zgoraj;
- argmin funkcija, ki vrne indeks elementa, ki ima minimalno ceno seštevka vektorjev s in v;
- min funkcija, ki sestavi vektorja v1 in v2 v v tako, da velja $v[i] = min\{v1[i], v2[i]\}$.



Slika 5.10: Računanje cene minimalnega vpetega drevesa

5.5.4 Računanje drevesa

Do zdaj smo zanemarili povezave minimalnega vpetega drevesa, računali smo le ceno minimalnega vpetega drevesa. Zapomniti si povezave je pri Primovem algoritmu relativno enostavno z uporabo pristopa z 2-terkami. Vsak vnos v matriki sosednosti A (in vektorju d) je 2-terka $A(u, v) = (W(u, v), u)$. Edina sprememba pri algoritmu je, da moramo shranjevati še vektor dimenzije $1 \times N$ π, v katerega shranimo vpeto drevo za vsa vozlišča v S.

Algoritem 5.5.34: Algebrska implementacija Primovega algoritma z računanjem drevesa

```
s=0;
cena=0;
s=Random(s iz V);
```

```
s(1)=Integer.MAX_VALUE;
d=A(1,:);
pi= NIL;
while s ni Integer.MAX_VALUE{
          u=argmin{s+v};
          s(u)=Integer.MAX_VALUE;
          cena=cena+d(u);
          pi (u)=p
          d=d min.+A(u,:);
}
```

Ker elementi v vektorju d in matriki A natanko ustrezajo posameznim povezavam, je očitno, da ta različica algoritma pravilno vrne povezave vpetega drevesa in skupno ceno teh povezav, ki je pravzaprav cena vpetega drevesa. Opazimo, da je shranjevanje povezav enostavnejše kot shranjevanje minimalnih poti pri prejšnjih algoritmih. Razlog je v tem, da vektor d hrani le povezave, medtem ko pri prejšnjih algoritmih shranjujemo cele poti. Drugi razlog je struktura algoritma, tukaj ne operiramo čez kak poseben polkolobar, zato ne potrebujemo kompleksne logike.

Poglavje 6

Testiranje algoritmov

6.1 Uvod v testiranje

V tem poglavju bomo testirali vse zgoraj opisane algebrske implementacije algoritmov. Vsakega od teh algoritmov smo v programskem jeziku Java sprogramirali na dva načina. Prvi način je algebrska implementacija algoritma, ki pri zapisu matrike uporablja standarni način zapisa matrike s poljem (tabelo), drugi pa je algebrska implementacija algoritma, ki pri zapisu matrike uporablja enega od zgoraj opisanih načinov zapisa redke matrike. Pri vsakem od sprogramiranih metod bo zraven tudi napisano, kateri način zapisa matrike uporablja. Vsi algoritmi so bili napisani ročno, brez uporabe knjižnic za računanje z matrikami ali pa za računanje z redkimi matrikami. Edini uporabljeni knjižnici sta bili java.util.LinkedList zaradi uporabe LinkedListov pri nekaterih metodah ter java.io.* zaradi branja matrik iz datoteke.

Sprogramirane metode so priložene na zgoščenki.

Namen naših testiranj algoritmov bo primerjava med standardnim zapisom matrike in zapisom matrike v enem od načinov zapisov redkih matrik. Naš cilj bo: za vsak algoritem ugotoviti, kateri zapis matrike in pri kateri testni matriki bi se nam bolj splačal uporabiti, to pomeni, da bi bil hitrejši. Glede na to, da pričakujemo, da se bo v večini primerov čas pri standardnih zapisih in zapisih redkih matrik spreminjal različno (pri standardnem zapisu se čas povečuje z dimenzijo matrike,

pri zapisu redkih matrik pa bi se moral s številom neničelnih elementov), bomo verjetno lahko določili število neničelnih elementov pri neki dimenziji matrike, kjer bo zapis z redko matriko še uporaben, pri večjem številu neničelnih elementov pa ne več, saj bo standardni zapis hitrejši.
Vse testne matrike so v datotekah zapisane po vrsticah, ena vrstica v datoteki pomeni eno vrstico v matriki. Pri vseh metodah smo matriko najprej prebrali v pomnilnik, potem pa izvajali nadaljnje izračune na njej, saj smo tako prihranili čas pri dostopu do elementov matrike.
Testni vektorji so v datotekah zapisani v eni vrstici, vrstica v datoteki je v bistvu kar vektor sam. Tudi vektor smo pri vseh metodah, ki računajo z vektorji, najprej prebrali v pomnilnik in nato do elementov vektorja dostopali iz pomnilnika in ne iz datoteke.

6.2 Množenje matrike z vektorjem

OPIS TESTA

Preverjali bomo zmogljivost množenja matrike z vektorjem, z različnimi zapisi redkih matrik in s standardnim zapisom matrike s poljem. Množenje matrike z vektorjem nas zanima, saj vsi nadaljni algoritmi v svojih izračunih uporabljajo predvsem množeje matrike z vektorjem, zato bo to ključna operacija, ki bo močno vplivala na učinkovitost ostalih metod.

OPIS UPORABLJENIH METOD

- Standardni zapis s poljem (AR) - metoda `Matrix.mnozenjeVektor`
- CSR zapis redke matrike (CSR) - metoda `Sparse.mnozenjeMatVecCSR`
- MSR zapis redke matrike (MSR) - metoda `Sparse.mnozenjeMatVecMSR`
- Zapis redke matrike s terkami (TUP) - metoda `Sparse.mnozenjeMatVecTER`

Vsako metodo bomo pograli 300000-krat in izračunali povprečni čas trajanja vsake metode, ki je zapisan v tabeli 6.1.

Čas je sicer odvisen od več dejavnikov (zmogljivosti računalnika...), ker pa bomo vse te metode merili na istem računalniku in pri enakih pogojih, lahko ta podatek vzamemo kot pokazatelj zmogljivosti metode, ne moremo pa trditi, da bi katerakoli od teh metod vsepovsod in v vseh pogojih delala toliko časa, kot je zapisano v tabeli 6.1.

`Matrix.mnozenjeVektor(int[][] matrika1, int[] vektor)`
Metoda opravi množenje matrike `matrika1` z vektorjem `vektor` (`matrika1 x vektor`). Množenje je implementirano po opisu množenja matrike z vektorjem v podpoglavju 4.2.

<u>Vhodni podatki:</u>

- 2D `matrika1` in 1D `vektor`, s katerim želimo množiti.

<u>Izhodni podatki:</u>

- vektor v, za katerega velja v= `matrika1 x vektor`.

`Sparse.mnozenjeMatVecCSR(int[] data, int[] ind, int[] poin, int[] x)`
Metoda opravi množenje matrike A in vektorja x, ko je matrika v CSR zapisu (Ax). Množenje je implementirano po algoritmu 3.4.1.

<u>Vhodni podatki:</u>

- `data`...polje z neničelnimi vrednostmi matrike po vrsti po vrsticah;
- `ind`...polje z indeksi stolpcev teh vrednosti `ind[i]` je vrstica za podatek `data[i]`;
- `poin`...polje, ki pove s katerimi od vrednosti v `data` se začne vrstica;
- `x` je vektor s katerim želimo množiti matriko.

<u>Izhodni podatki:</u>

- vektor `y`, za katerega velja $y = Ax$,

`Sparse.mnozenjeMatVecMSR(int[] data, int[] ind, int[] x)`
Metoda opravi množenje matrike A in vektorja x, ko je matrika v MSR zapisu (Ax). Množenje je implementirano po algoritmu 3.4.5.

<u>Vhodni podatki:</u>

- `data, ind` so podatki o matriki A, predstavljeni v MSR načinu;
- `x` vektor, s katerim množimo matriko.

Izhodni podatki

- vektor `y`, za katerega velja $y = Ax$.

`Sparse.mnozenjeMatVecTER(int[] data, int[] col, int[] row, int[] x)`
Metoda opravi množenje matrike A in vektorja x, ko je matrika v zapisu s terkami (Ax). Množenje je implementirano po algoritmu 3.4.7.
Vhodni podatki:

- `data, col, row` so podatki o matriki A, predstavljeni v načinu s terkami;
- `x` vektor, s katerim množimo matriko.

Izhodni podatki:

- vektor `y`, za katerega velja $y = Ax$.

TESTNE MATRIKE IN VEKTORJI
Pri testiranju bomo uporabili spodaj zapisane matrike in vektorje; matrike imajo imena testX, kjer je X število, vektorji pa vectorY, kjer je Y število.

- **test1** je matrika dimenzije 200 x 200, s 3776 neničelnimi vrednostmi od 1 do 5.
- **test2** je matrika dimenzije 200 x 200, z 10068 neničelnimi vrednostmi od 1 do 5.
- **test3** je matrika dimenzije 100 x 100, z 2100 neničelnimi vrednostmi od 1 do 5.
- **test4** je matrika dimenzije 100 x 100, s 763 neničelnimi vrednostmi od 1 do 5.
- **test5** je matrika dimenzije 300 x 300, s 17413 neničelnimi vrednostmi od 1 do 5.

- **test6** je matrika dimenzije 300 x 300, z 2365 neničelnimi vrednostmi od 1 do 5.
- **vector1** je vektor dimenzije 1 x 100, z 18 neničelnimi vrednostmi od 1 do 5.
- **vector2** je vektor dimenzije 1 x 200, s 30 neničelnimi vrednostmi od 1 do 5.
- **vector3** je vektor dimenzije 1 x 300, z 69 neničelnimi vrednostmi od 1 do 5.

REZULTATI

$MATRIKA$	$test1$	$test2$	$test3$	$test4$	$test5$	$test6$
$VEKTOR$	$vector2$	$vector2$	$vector1$	$vector1$	$vector3$	$vector3$
AR	44864	44930	16900	16917	99463	99500
CSR	14980	38566	8819	5282	65960	13080
MSR	18585	49008	10719	4760	84116	13466
TUP	18105	47948	10316	4362	66060	13255

Tabela 6.1: Rezultati testiranj algoritmov za množenje matrike z vektorjem

*Opomba: Časi v tabeli so zapisani v nanosekundah.
Tabelo moramo brati navpično, primer: pri matriki test1 in vektorju vector2 je metoda AR delovala 44864 ns, metoda CSR 14980 ns in tako dalje.

OPAŽANJA

Kot pričakovano, je pri metodi s standardnim zapisom matrike čas delovanja odvisen od dimenzije matrike, ne pa od tega, koliko imamo neničelnih elementov, saj metoda vedno preračuna vsa polja, ne glede na vrednost. Vsem ostalim metodam se čas delovanja povečuje s številom neničelnih elementov, kar se je v našem primeru izkazalo za bolje, saj smo imeli za testne matrike redke matrike, ki imajo veliko ničel. Med algoritmi, ki pri izračunih upoštevajo število neničelnih elementov, se je najbolje odrezal zapis v CSR obliki. Pri zelo majhnem številu neničelnih elementov sta bila MSR in način s terkami malenkost hitrejša, a nas bodo v splošnem bolj zanimale večje matrike. MSR način sicer ne zaostaja prav dosti za CSR načinom zapisa matrike, a v naših algoritmih MSR zapis ne bo prišel v poštev, saj MSR način predvideva neničelno diagonalo, kar pa pri matriki cen ni res. Najbolj

učinkovit pri množenju matrike z vektorjem je pri naših testnih matrikah torej način zapisa matrike s CSR zapisom.
Ker se je zapis redke matrike v CSR obliki najbolj obnesel že pri množenju matrike z vektorjem (kar je osnovna operacija za vse nadaljnje algoritme), bomo pri ostalih algoritmih primerjali samo ta zapis z osnovnim zapisom matrike s poljem. Če bomo pri katerem algoritmu ugotovili, da bi bil lahko kateri od zapisov redkih matrik zaradi načina pregledovanja neničelnih elementov boljši, bomo v test vključili še ta način zapisa.
Pričakujemo lahko, da bo pri nekaterih algoritmih vseeno hitrejši algoritem s standardnim zapisom matrike s poljem, ne glede na to, da je osnovno množenje matrike z vektorjem v tem zapisu počasnejše, saj pri nekaterih algoritmih za zapis s CSR-jem potrebujemo veliko dodatnih primerjav in pretvorb, ki jih pri navadnem zapisu ne potrebujemo. To lahko ponekod poslabša zmogljivost algoritma.

6.3 Iskanje v širino (BFS algoritem)

OPIS TESTA

Preverjali bomo zmogljivost algoritma BFS, ki je teoretično opisan v podpoglavju 5.1. Algoritem bomo implementirali s tremi različnimi zapisi redkih matrik, kar pomeni, da bomo sprogramirali tri različne metode, ki prikazujejo delovanje algoritma in vsaka uporablja drug zapis matrike.

Algoritem 6.3.36: BFS algoritem

```
B=A;
for i:=1 to st stopenj step 1 {
        B=B OR.AND A;
}
```

- Standardni zapis s poljem (AR) - metoda `Matrix.BFS`
- CSR zapis redke matrike (CSR) - metoda `Sparse.BFS_CSR_matrix`

- Zapis redke matrike s terkami (TUP)- `metoda Sparse.BFS_TER`

Algoritem bo iskal sosede v N-1 korakih, kjer je N število vozlišč v grafu. Vsako metodo bomo pognali 1000-krat in izračunali povprečni čas, ta čas je zapisan v tabeli 6.2.

OPIS UPORABLJENIH METOD

`Matrix.BFS(int[][] matrika,int k)`

Metoda simulira algoritem za iskanje v širino (BFS), implementirana je po algoritmu 6.3.35. Pri delovanju uporablja množenje matrik v polkolobarju (AND, OR), kjer so matrike zapisane v standardnem zapisu s poljem.

<u>Vhodni podatki:</u>

- sosedna matrika `matrika`;
- `k` je število stopenj, do koder želimo preiskovati sosede v širino.

<u>Izhodni podatki</u>

- 2D matrika `C`, za katero velja: če `C[i][j] =1`, potem iz vozlišča i v j lahko pridemo v k. korakih

Pomožne metode:

`1. Matrix.mnozenjeBinOR_AND(int[][] matrika2, int[][] matrika1)`

Metoda izvede množenje 2 matrik, ki so zapisane v standardnem zapisu matrike s poljem v kolobarju (OR,AND). Primer takega množenja je prikazan na sliki 5.2.

<u>Vhodni podatki:</u>

- `matrika1` in `matrika2` enakih dimenzij sta 2D matriki, ki ju želimo zmnožiti

<u>Izhodni podatki</u>

- Rezultat je 2D matrika iste dimenzije kot `matrika1` ali `matrika2`, taka, da velja `rezultat=matrika1 OR.AND matrika2`.

`Sparse.BFS_CSR_matrix(int[]data, int[] ind, int[] poin, int k)`

Metoda simulira algoritem za iskanje v širino (BFS), implementirana je po algoritmu, opisanem zgoraj. Pri delovanju uporablja množenje matrik v polkolobarju (AND, OR), kjer so matrike zapisane v CSR zapisu redkih matrik.
Vhodni podatki:

- `data, ind` in `poin` so podatki o matriki A v CSR načinu zapisa;
- `k` je število stopenj, do koder nas zanimajo sosedi.

Izhodni podatki:

- matrika 2D `C`, ki nam pove, do katerih vozlišč pridemo v k korakih. Če je v `C[i][j]=1`, lahko iz i v j pridemo v k korakih.

Pomožne metode:

1. `Sparse.mnozenjeMatCSR_OR_AND(int[] data1, int[] ind1, int[] poin1, int[] data2, int[] ind2, int[] poin2)`

Metoda izvede množenje 2 matrik, ki so zapisane v CSR zapisu v kolobarju (OR,AND). Primer takega množenja je prikazan na sliki 5.2.
Vhodni podatki:

- `data1, ind1, poin1` so podatki o matriki A, predstavljeni v načinu CSR;
- `data2, ind2, poin2` so podatki o matriki B, predstavljeni v načinu CSR.

Izhodni podatki:

- matrika `C` dimenzije A ali B, za katero velja C=A OR.AND B.

2. `Sparse.fromNormalToCSR(int[][] matrika)`

Metoda izvede pretvorbo iz standardnega zapisa matrike v CSR zapis redke matrike.
Vhodni podatki:

- Standardno zapisana matrika z 2D poljem `matrika`.

Izhodni podatki:

- `LinkedList`, ki po vrsti vsebuje polja `data, ind` in `poin` za vhodno matriko.

`Sparse.BFS_TER(int[]data, int[] row, int[] col, int k,int dim)`
Metoda simulira algoritem za iskanje v širino (BFS), implementirana je po algoritmu, opisanem zgoraj. Pri delovanju uporablja množenje matrik v polkolobarju (AND, OR),kjer so matrike zapisane v zapisu redkih matrik s terkami.
Vhodni podatki:

- `data row` in `col` so podatki o matriki A v načinu zapisa s terkami;
- `k` je število "stopenj", do koder nas zanimajo sosedi;
- `dim` je število vozlišč v osnovnem grafu.

Izhodni podatki:

- matrika 2D `C`, ki nam pove, do katerih vozlišč pridemo v k korakih. Če je v `C[i][j]=1`, lahko iz i v j pridemo v `k` korakih.

Pomožne metode:
`mnozenjeMatTER_OR_AND(int[] data, int[] row, int[] col, int data1[], int[] row1, int[] col1,int dim)`
Metoda izvede množenje 2 matrik, ki so zapisane v zapisu s terkami v kolobarju (OR,AND).
Vhodni podatki:

- `data, col, row` so podatki o matriki A, predstavljeni v načinu s terkami;
- `data1, col1, row1` so podatki o matriki B, predstavljeni v načinu s terkami;
- `dimenzija` je število vozlišč v osnovnem grafu.

Izhodni podatki:

- matrika `C`, za katero velja C=A OR.AND B.

TESTNE MATRIKE

- **test7** je matrika dimenzije 200 x 200, z 2378 enicami.

- **test8** je matrika dimenzije 200 x 200, s 6218 enicami.
- **test9** je matrika dimenzije 100 x 100, s 1632 enicami.
- **test10** je matrika dimenzije 100 x 100, s 687 enicami.
- **test11** je matrika dimenzije 100 x 100, z 49 enicami.

REZULTATI

$MATRIKA$	$test7$	$test8$	$test9$	$test10$	$test11$
AR	4,77	4,58	0,23	0,23	4,87
CSR	6,05	6,47	0,41	3,84	0,11
TUP	21,75	60,24	2,04	0,89	0,04

Tabela 6.2: Rezultati testiranj algoritma BFS

*Opomba: Časi v tabeli so zapisani v $10^9 ns$.
Tabelo moramo brati navpično, primer: pri matriki test7 je metoda AR delovala 4,77 10^9 ns in tako dalje.

OPAŽANJA

Opazimo, da še vedno drži, da se pri standardnem zapisu matrike čas izvajanja povečuje z dimenzijo matrike in ne s številom neničelnih elementov, kot pri zapisu matrik z zapisi za redke matrike. Zaradi kompleksnosti zapisa redke matrike in posledično kompleksnosti izračunov s tem zapisom matrike, se ta zapis izkaže za slabšega. V zadnjem primeru, ko je neničelnih elementov res malo v primerjavi s celotnim številom elementov, sta oba načina zapisa matrike z redkimi matrikami boljša, saj morata pregledati res malo elementov v primerjavi s standardnim zapisom matrike, ki vedno pregleda vse možne. Zapis redke matrike v načinu s terkami je boljši od standardnega zapisa in CSR načina zapisa, razen v zadnjih dveh primerih (test10 in test11), ko je pri testu 10 boljši od CSR načina zapisa in pri testu11 boljši od obeh. Zapis s terkami se v teh primerih izkaže za boljšega, ker je dostop do elementov pri tem zapisu enostavnejši kot zapis CSR, pri katerem direktno dostopamo le do stolpca elementa, vrstico pa je treba iskati. Pri zapisu s terkami imamo podatka o vrstici in stolpcu vedno znana.

Opazimo, da obstajajo testne matrike, za katere je boljši standardni zapis matrike, in da obstajajo tudi testne matrike, za katere je boljši zapis CSR. Glede na to, da na zmogljivost metode s standardnim zapisom matrike vpliva le dimenzija matrike, na zmogljivost metode z zapisom CSR pa vpliva število neničelnih elementov, na nobeno od teh metod pa ne vpliva lega (vrstica in stolpec) neničelnih elementov, moramo najti matriko z določenim številom neničelnih elementom, pri kateri je še boljši zapis matrike CSR; ko pa dodamo nekaj neničelnih elementov, pa je že boljši standardni zapis matrike. Glede na to, da tudi pri 1000 ponovitvah ne dobimo vsakič identičnega časa izvajanja metode, ne bomo mogli določiti te meje na razliko enega neničelnega elementa, zato smo tudi rekli, da bomo vzeli približno mejo z možnimi odstopanji v nekaj več ali manj neničelnih elementov.

Želimo torej najti pribižno mejo v številu neničelnih elementov pri matriki dimenzije 200 x 200, kjer je metoda z zapisom redke matrike CSR še boljša kot pa metoda, ki uporablja standardni zapis matrike.

Testa smo se lotili tako, da smo zgenerirali testno matriko dimenzije 200 x 200 z 200 neničelnimi elementi, za katero smo ugotovili, da je metoda s CSR zapisom matrike boljša, nato smo dodajali neničelne elemente na poljubna mesta (kot smo ugotovili, mesto elementov ni pomembno), dokler nismo prišli do števila neničelnih elementov v matriki, ko je metoda s standardnim zapisom matrike postala hitrejša od tiste s CSR zapisom. Ta meja je pri matriki dimenzije 200 x 200 približno pri 205 neničelnih elementih (matrika, ki je med testiranjem nastala in je na koncu pokazala približno mejo, ko je metoda s CSR načinom zapisa še hitrejša od metode s standardnim zapisom, je test_adj).

6.4 Iskanje minimalnih poti

OPIS TESTA

Preverjali bomo zmogljivost algoritma iskanje minimalnih poti, ki je teoretično opisan v podpoglavju 5.2. Algoritem bomo implementirali z dvema metodama, ki bosta uporabljali različna zapisa matrik. Spodnji metodi sta sprogramirani po metodi 5.2.13:

- Standardni zapis s poljem (AR) - metoda `Matrix.iskanjeNajkrajsihPoti`
- CSR zapis redke matrike (CSR) - metoda `Sparse.iskanjeNajkrajsihPotiCSR`

Vsako metodo bomo pognali 1000-krat in izračunali povprečni čas, ki je zapisan v tabeli 6.3.

OPIS UPORABLJENIH METOD

`Matrix.iskanjeNajkrajsihPoti(int[][] matrika)`

Z metodo iščemo cene minimalnih poti za vse možne pare vozlišč grafa, matrike, s katerimi se opravijo izračuni, so zapisane s standardnim zapisom s poljem. Algoritem je implementiran po psevdokodi 5.2.13.

Vhodni podatki:

- matrika cene `matrika`.

Izhodni podatki:

- matrika, ki vsebuje cene minimalnih poti za vse pare vozlišč grafa.

Pomožne metode

`1. Matrix.mnozenjeMinPlus(int[][] matrika)`

Metoda izvede kvadriranje matrike v polkolobarju (min,+), pri čemer je matrika v standardnem zapisu s poljem. Primer računanja v polkolobarju (min,+) in algoritem, po katerem je implementirana metoda je v podpoglavju 5.2.

Vhodni podatki:

- 2-dimenzionalna matrika `matrika`.

Izhodni podatki:

- 2-dimenzionalna matrika `C`, za katero velja `C=matrika x matrika` v kolobarju (min,+).

`Sparse.iskanjeNajkrajsihPotiCSR(int[]data, int[] ind, int[] poin)`

Metoda simulira delovanje algoritma minimalne poti, pri čemer so matrike, ki so uporabljene za izračune zapisane v CSR zapisu. Algoritem je implementiran po psevdokodi 5.2.13.

Vhodni podatki:

- `data, ind, poin` so podatki o matriki cen A podani v CSR načinu zapisa.

Izhodni podatki:

- `LinkedList` s podatki `data, ind, poin` v tem vrstnem redu matrike C, za katero velja, da je v `C[i][j]` minimalna cena poti med vozliščema i in j .

Pomožne metode

1. `Sparse.fromNormaltoCSR_nesk(int[][] matrika)`

Metoda izvede pretvorbo iz normalne matrike v CSR; kjer neskončne vrednosti obravnavamo kot 0 (shranjujemo vrednosti različne od ∞).

Vhodni podatki:

- standardno zapisana matrika z 2-dimenzionalnim poljem.

Izhodni podatki:

- `LinkedList`, ki po vrsti vsebuje polja `data, ind, poin` za vhodno matriko.

2. `Sparse.mnozenjeMatMinPlus_CSR(int[]data,int[] ind, int[] poin)`

Metoda izvede množenje 2 matrik (enakih dimenzij) v kolobarju (min,+), pri čemer je matrika zapisana s CSR zapisom. Primer računanja v polkolobarju (min,+) in algoritem, po katerem je implementirana metoda, je v podpoglavju 5.2.

Vhodni podatki:

- `data, ind, poin` so podatki o matriki A v CSR načinu zapisa, kjer je CSR način prilagojen tako, da hrani podatke različne od neskončno.

Izhodni podatki:

- matrika `C`, za katero velja $C = A\ min. +\ A$.

TESTNE MATRIKE

- **test13** je matrika dimenzije 200 x 200, s 3766 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.
- **test14** je matrika dimenzije 200 x 200, z 10015 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.

- **test15** je matrika dimenzije 100 x 100, z 2043 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.
- **test16** je matrika dimenzije 100 x 100, s 1080 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.
- **test17** je matrika dimenzije 100 x 100, s 86 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.

REZULTATI

$MATRIKA$	$test13$	$test14$	$test15$	$test16$	$test17$
AR	17,64	16,93	1,58	1,76	18,04
CSR	32,98	32,78	3,74	3,83	5,73

Tabela 6.3: Rezultati testiranj algoritma iskanja minimalnih poti

*Opomba: Časi v tabeli so zapisani v $10^7 ns$.
Tabelo moramo brati navpično, primer: pri matriki test13 je metoda AR delovala 17,64 10^7 ns in tako dalje.

OPAŽANJA

Pri metodi, ki uporablja standardni zapis matrike, se čas izvajanja povečuje z dimenzijo matrike in ne s številom neničelnih elementov kot pri zapisu CSR. Zapis s CSR se v večini primerov izkaže za slabšega, razen v zadnjem primeru, ko je neničelnih elementov res malo v primerjavi s celotnim številom elementov, ki jih pregleda algoritem pri standardnem zapisu matrike.
Enako kot pri BFS algoritmu ugotovimo, da očitno obstajajo testne matrike (tiste, ki imajo zelo malo število neničelnih elementov), pri katerih je metoda s CSR zapisom boljša od metode s standardnim zapisom matrike. Kot smo že ugotovili, lega neničelnega elementa ne vpliva na zmogljivost metode, pomembno je le število neničelnih elementov
Kakšna pa bi tu bila meja, ko je CSR zapis še boljši kot standardni?
Začeli bomo z matriko dimenzije 200 x 200, ki ima 200 neničelnih (in različnih od ∞) elementov. Opazimo, da je tu metoda, ki uporablja CSR zapis matrike,

boljša. Matriki na poljubna mesta (ki niso pomembna) dodajamo neničelne (in različne od ∞) elemente, dokler ni metoda s standardnim zapisom boljša.
Pri matriki dimenzije 200 x 200 je meja približno pri 230 elementih, različnih od ∞ (mejo smo dosegli z nastankom matrike test_cost).

6.5 Bellman-Fordov algoritem

OPIS TESTA
Preverjali bomo zmogljivost Bellman-Fordovega algoritma. Implementirali bomo algebrsko različico algoritma z dvema metodama, ki uporabljata različna zapisa matrik in bomo za vsakega od teh zapisov preverjali 2 načina implementacije algoritma, ki sta opisana v podpoglavju 5.3. Oba načina sta implementirana po psevdokodi, ki je zapisana v že prej omenjenem podpoglavju.
V prvem delu testiranja bomo testirali naslednje metode, pri katerih si shranjujemo samo **ceno** minimalne poti za poljubno vozlišče, ne pa tudi poti same. Brez izgube za splošnost bomo v našem primeru vedno iskali minimalne poti za prvo vozlišče.
Implementirane metode:

- standardni zapis s poljem (AR1) - `Matrix.BellmanFord_prvaVerzija_normal`
- standardni zapis s poljem (AR2) - `Matrix.BellmanFord_drugaVerzija_normal`
- CSR zapis redke matrike (CSR1) - `Sparse.BellmanFord_prvaVerzija`
- CSR zapis redke matrike (CSR2) - `Sparse.BellmanFord_drugaVerzija`

Vsako metodo bomo pognali 100-krat in izračunali povprečni čas, ki je zapisan v tabeli 6.4.

OPIS UPORABLJENIH METOD
`Matrix.BellmanFord_prvaVerzija_normal(int[][] matrika, int vozlisce)`
Metoda simulira Bellman-Fordov algoritem, implementiran tako, da računamo samo cene minimalnih poti od želenega vozlišča do vseh ostalih in ne vseh cen minimalnih poti za vse možne pare vozlišč. Implementacija je narejena na prvi način, po algoritmu 5.3.17. Metoda uporablja standardni zapis matrike s poljem.

Vhodni podatki:

- 2-dimenzionalna matrika cen `matrika`;
- indeks vozlišča `vozlisce` za katerega nas zanimajo minimalne poti (1-N).

Izhodni podatki:

- vektor `d`, ki vsebuje cene minimalnih poti od želenega vozlišča do vseh ostalih `d[i]=a` pomeni, da je a cena minimalne poti od vozlišča `vozlisce` do i.

Pomožne metode
`1. Matrix.mnozenjeMINPLUSZDesne(int[][] matrika, int[] vektor)`
Metoda izvede množenje vektorja z matriko (xA) v polkolobarju (min,+), kjer metoda uporablja standardni način zapisa matrike s poljem. Primer računanja v polkolobarju (min,+) je v podpoglavju 5.2.
Vhodni podatki:

- 2-dimenzionalna matrika `matrika`;
- vektor `vektor`, s katerim želimo množiti.

Izhodni podatki:

- vektor `rezultat`, za katerega velja $rezultat = vektor\ min. +\ matrika$.

`Matrix.BellmanFord_drugaVerzija_normal(int[][] matrika, int vozlisce)`
Metoda simulira Bellman-Fordov algoritem, implementiran tako, da izračuna vnaprej cene minimalnih poti za vse pare vozlišč grafa in na koncu izbere vrstico, ki vsebuje minimalne poti za želeno vozlišče. Ta način je v podpoglavju 5.3. opisan pod drugo verzijo, implementacija je narejena po algoritmu 5.3.19. Metoda uporablja zapis matrike v standardnem zapisu s poljem.
Vhodni podatki:

- 2-dimenzionalna matrika cen `matrika`;
- indeks vozlišča `vozlisce` za katerega nas zanimajo minimalne poti (1-N).

Izhodni podatki:

- vektor `d`, ki vsebuje cene minimalnih poti od želenega vozlišča do vseh ostalih `d[i]=a`, pomeni, da je a cena minimalne poti od vozlišča `vozlisce` do i.

Pomožne metode

1. `Matrix.iskanjeNajkrajsihPoti(int[][] matrika)`
2. `Matrix.mnozenjeMINPLUSZDesne(int[][] matrika, int vozlisce)`

```
Sparse.BellmanFord_prvaVerzija(int[]data,int[] ind, int[] poin,
int vozlisce)
```

Metoda simulira Bellman-Fordov algoritem, implementiran tako, da računamo samo cene minimalnih poti od želenega vozlišča do vseh ostalih in ne vseh cen minimalnih poti za vse možne pare vozlišč. Implementacija je narejena po algoritmu 5.3.19, metoda uporablja CSR zapis matrike.

Vhodni podatki:

- `data, ind, poin` so podatki o matriki cen A podai s CSR zapisom;
- `vozlisce` je indeks vozlišča (1-N), za katerega nas zanimajo minimalne poti.

Izhodni podatki:

- vektor `d`, za katerega velja, da je v `d[i]` cena minimalne poti od vozlišča v spremenljivki `vozlisce` do i.

Pomožne metode

```
1. Sparse.mnozenjeMatvec_MinPlusZDesne(int[]data,int[] ind,
int[] poin, int[] x)
```

Metoda izvede množenje vektorja in matrike xA v kolobarju (min,+), kjer metoda uporablja CSR zapis matrike. Primer računanja v polkolobarju (min,+) je v podpoglavju 5.2.

Vhodni podatki:

- `data, ind, poin` so podatki o matriki A, podani v CSR zapisu;
- `x` je vektor, s katerim želimo množiti.

Izhodni podatki:

- vektor `y`, za katerega velja $y = x\ min. +\ A$.

`Sparse.BellmanFord_drugaVerzija(int[]data,int[] ind, int[] poin, int vozlisce)`
Metoda simulira Bellman-Fordov algoritem, implementiran tako, da izračuna vnaprej cene minimalnih poti za vse pare vozlišč grafa in na koncu izbere vrstico, ki vsebuje minimalne poti za želeno vozlišče. Metoda je implementirana po algoritmu 5.3.19.
Vhodni podatki:

- `data, ind, poin` so podatki o matriki cen A, podani v CSR zapisu;
- `vozlisce` je indeks vozlišča (1-N), za katerega nas zanimajo minimalne poti.

Izhodni podatki:

- vektor `d`, za katerega velja, da je v `d[i]` cena minimalne poti od vozlišča v spremenljivki `vozlisce` do i.

Pomožne metode:
1. `Sparse.iskanjeNajkrajsihPotiCSR(int[] data, int[] ind, int[] poin,int vozlisce)`
2. `Sparse.mnozenjeMatvec_MinPlusZDesne(int[] data, int[] ind, int[] poin, int x)`

TESTNE MATRIKE

- **test13** je matrika dimenzije 200 x 200, s 3766 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.
- **test14** je matrika dimenzije 200 x 200, z 10015 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.
- **test15** je matrika dimenzije 100 x 100, z 2043 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.

- **test16** je matrika dimenzije 100 x 100, s 1080 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.
- **test17** je matrika dimenzije 200 x 200, s 86 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.

REZULTATI

$MATRIKA$	$test13$	$test14$	$test15$	$test16$	$test17$
AR1	28,31	26,91	2,91	3,75	26,28
AR2	178,46	176,20	15,80	17,67	180,36
CSR1	50,82	61,85	7,32	6,71	42,90
CSR2	415,69	411,17	46,86	47,67	57,85

Tabela 6.4: Rezultati testiranj Bellman-Fordovega algoritma brez shranjevanja poti

*Opomba: Časi v tabeli so zapisani v $10^6 ns$.
Tabelo moramo brati navpično, primer: pri matriki test13 je metoda AR delovala 28,31 10^6 ns in tako dalje.

OPAŽANJA

Prva verzija Bellman-Fordovega algoritma je počasnejša, če matriko shranjujemo v CSR obliki, saj je množenje v x A (vektorja z matriko iz leve strani) počasnejše v tej obliki, kot pa v standardnem zapisu matrike. Pri prvi verziji se bolj splača uporabiti standardni zapis matrike, tudi, če je neničelnih elementov zelo malo, kar lahko vidimo pri matriki test17. Druga verzija Bellman-Fordovega algoritma je sicer pri matriki v CSR obliki še vedno počasnejša v večini primerov, a pri test17 vidimo, da se pri malem številu neničelnih elementov vseeno bolj splača.
Pri prvi verziji števila elementov, različnih od ∞, ko se nam bolj splača CSR zapis sploh ne bomo iskali meje, ko bi bil CSR zapis boljši, saj je metoda v vseh primerih zaostajala za drugo s standardnim zapisom matrike, tudi pri test17, ko je takih elementov zelo malo.
Pri drugi verziji implementacije algoritma bomo mejno število elementov, različnih od ∞, poiskali na enak način, kot smo to storili pri zgornjih primerih. Začnemo z

matriko dimenzije 200 x 200, ki ima 200 elementov, različnih od ∞. Opazimo, da je v tem primeru hitrejša metoda, ki za zapis matrike uporablja standardni zapis s poljem. Matriki odvzemamo elemente, različne od ∞, tako da jih nadomestimo z elementom ∞. Mejno število elementov, različnih od ∞, ko se nam CSR zapis matrike pri matriki dimenzije 200 x 200 splača, dobimo nekje pri 170 elementih, različnih od ∞, (mejo smo dosegli z nastankom matrike matriki test_cost1). Taka matrika nikoli ne more biti sosedna matrika povezanega grafa, saj bi v tem primeru potrebovali vsaj 199 povezav.
Zgornje testne matrike so bile generirane naključno, zato pripadajoči grafi niso nujno povezani. Pri matrikah, ki jih dobimo iz povezanega grafa, pa druga verzija metode z zapisom matrike CSR delujejo bolje, a v večini primerov še vedno slabše, kot če zapišemo matriko na standarden način. Testiranje za ugotavljanje mejnega števila elementov, ko bi se nam bolj splačala uporaba metode s CSR zapisom matrike, sedaj ponovimo na sosedni matriki povezanega grafa. Začnemo z matriko dimenzije 200 x 200, z 200 elementi, različnimi od ∞, ki je sosedna matrika povezanega grafa, in ugotovimo, da je v tem primeru metoda s CSR zapisom matrike hitrejša. Dodajamo elemente, različne od ∞, dokler metoda s standardnim zapisom matrike ne postane hitrejša. Mejno število, ko se nam zapis CSR še splača, je pri matriki dimenzije 200 x 200 približno pri 252 elementih, različnih od ∞.
Po ugotovitvi, da rezultati, dobljeni pri zgornjih testnih matrikah, ki niso bile nujno sosedne matrike povezanih grafov, morda niso pokazatelji prave slike, smo ponovili testiranja na naslednjih matrikah, ki so sosedne matrike povezanih grafov

- **test18** je matrika dimenzije 200 x 200, s 4860 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.
- **test19** je matrika dimenzije 100 x 100, s 1863 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.
- **test20** je matrika dimenzije 300 x 300, z 9750 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.

Vsako metodo smo pognali 100-krat in izračunali povprečni čas, ki je zapisan v tabeli 6.5.

$MATRIKA$	$test18$	$test19$	$test20$
AR1	40,12	5,25	156,12
AR2	179,88	17,06	1099,89
CSR1	68,03	7,45	174,33
CSR2	414,89	47,64	1553,40

Tabela 6.5: Rezultati testiranj Bellman-Fordovega algoritma na povezanih grafih

*Opomba: Časi v tabeli so zapisani v $10^6 ns$.

OPAŽANJA

Slika tudi na povezanih grafih ni bistveno drugačna, še vedno se nam v večini primerov bolj splača uporabiti standardni zapis matrike. Prva verzija je pri standardnem zapisu matrike hitrejša kot druga, pri CSR zapisu pa je druga verzija algoritma hitrejša kot prva.

DRUGI DEL TESTA

V drugem delu testiranja bomo za ilustracijo testirali prvo verzijo Bellman-Fordovega algoritma s standardnim zapisom matrike, ki vrača ceno minimalnih poti do drugih vozlišč iz enega vozlišča, in tudi polje, iz katerega rekonstruiramo pot samo. Prvo verzijo bomo prikazali, ker se je v prejšnjem testiranju izkazalo, da je hitrejša od druge. Glede na to, da shranjevanje poti lahko le upočasni metodo, ne pričakujemo, da bi lahko s shranjevanjem poti postala druga verzija boljša od prve. Prav tako smo že v prvem testiranju ugotovili, da je metoda z zapisom CSR veliko slabša, zato ne pričakujemo, da bi se s shranjevanjem poti sedaj slika obrnila. Zaradi tega implementacije z izračunavanjem poti sploh ne bomo delali z zapisom matrike CSR. Brez izgube za splošnost bomo v našem primeru vedno iskali minimalne poti za prvo vozlišče.

- Standardni zapis s poljem (AR) - `Matrix.BellmanFord_prvaVerzija_normal_path`

Metodo bomo pognali 100-krat in izračunali povprečni čas, ki je zapisan v tabeli 6.6. Teste bomo opravili na sosednih matrikah povezanih grafov (test18, test19 in test20).

OPIS UPORABLJENIH METOD

`Matrix.BellmanFord_prvaVerzija_normal_path(int[][] matrika, int vozlisce)`
Metoda simulira Bellman-Fordov algoritem, implementiran po prvi verziji, ki vrača tudi pot in ne samo ceno minimalnih poti. Metoda pri zapisu matrik uporablja standardni zapis matrike s poljem.
Vhodni podatki:

- 2-dimenzionalna matrika cen `matrika`;
- indeks vozlišča, za katerega nas zanimajo minimalne poti.

Izhodni podatki:

- `LinkedList`, ki vsebuje vektor minimalnih poti za določeno vozlišče in vektor predhodnih vozlišč, preko katerih smo prišli do minimalne cene poti.

Pomožne metode
`1. Matrix.mnozenjeMINPLUSZDesne(int[][] matrika, int[] x)`

$MATRIKA$	$test18$	$test19$	$test20$
AR	44,11	5,45	158,12

Tabela 6.6: Rezultati testiranj Bellman-Fordovega algoritma, ki vrača tudi minimalno pot

*Opomba: Časi v tabeli so zapisani v $10^6 ns$.
OPAŽANJA
V primerjavi z zgornjo verzijo algoritma, ki ne vrača tudi minimalne poti, ampak samo ceno le-te, opazimo, da dodatno shranjevanje predhodnih vozlišč ne porabi opazno veliko časa, saj je predhodnik direktno dostopen, zato shranjevanje le-tega predstavlja le eno operacijo shranjevanja elementa v polje.

6.6 Floyd-Warshallov algoritem

OPIS TESTA

Preverjali bomo zmogljivost Floyd-Warshallovega algoritma, implementirali bomo algebrsko različico Floyd-Warshallovega algoritma z dvema različnima zapisoma matrik. V ta namen bomo preizkušali dve metodi, obe vrneta cene minimalnih poti med vsemi pari vozlišč v grafu in indekse predhodnikov na tej poti, le da ena za zapis matrike uporablja standardni zapis, druga pa CSR zapis. Obe metodi sta implementirani po psevdokodi 5.4.23. Tudi pot v obeh metodah se izračunava, kot je opisano v podpoglavju 5.4. V primeru, da je indeks predhodnika 0, to pomeni, da je povezava direktna že v osnovnem grafu.
Implementirani metodi:

- standardni zapis s poljem (AR) - metoda `Matrix.FloydWarshall_normal`
- CSR zapis redke matrike (CSR) - metoda `Sparse.FloydWarshall`

Metodi bomo pognali 10000-krat in izračunali povprečni čas, ki je zapisan v tabeli 6.7.

OPIS UPORABLJENIH METOD

`Matrix.FloydWarshall_normal(int[][] ma)`

Metoda simulira delovanje algoritma Floyd Warshall. Uporablja standardni zapis matrike s poljem. Implementirana je po algoritmu 5.4.23.

Vhodni podatki:

- 2-dimenzionalna matrika `ma`, ki je matrika cen grafa, za katerega nas zanimajo minimalne poti med vozlišči.

Izhodni podatki:

- 3-dimenzonalna matrika - `rezultat`, ki vsebuje cene minimalnih poti v `rezultat[][][1]` in indekse predhodnih vozlišč, preko katerih smo prišli do minimalne cene poti v `rezultat[][][0]`.

`Sparse.FloydWarshall(int[] data, int[] ind, int[] poin)`

Metoda simulira delovanje Floyd-Warshallovega algoritma. Metoda uporablja CSR zapis matrike, implementirana je po algoritmu 5.4.23.

Vhodni podatki:

- `data, ind, poin` so podatki o matriki cen A, podani v CSR zapisu.

Izhodni podatki:

- 3-dimenzionalna matrika - `rezultat`, ki vsebuje cene minimalnih poti med vsemi pari vozlišč in predhodnike na tej minimalni poti.

Pomožne metode:

`1. Sparse.mnozenjeMatMinPLus_CSR_postop`
`(int[]data,int[] ind, int[] poin, int stop,int[] pred)`

Metoda izvaja "kvadriranje" dela matrike v kolobarju (min,+) $((D(:, stop)\ min. + D(stop, :))$.

Vhodni podatki:

- `data, ind, poin` so podatki o matriki A, podani v CSR zapisu;
- `stop` je število, do katerega želimo množiti stolpce in vrstice matrike;
- `pred` je tabela trenutnih predhodnikov, preko katerih smo prišli do trenutnih minimalnih poti.

Izhodni podatki:

- 3D matrika - `rezultat`, ki vsebuje cene minimalnih poti med vsemi pari vozlišč in predhodnike na tej minimalni poti.

`2. fromNormaltoCSR_nesk3D(int[][][] matrix)`

Metoda, ki iz standardne oblike zapisa matrike v 3D tabeli z dodanimi predniki pretvori v CSR zapis (kjer se vrednosti ∞ obravnavajo kot 0).

Vhodni podatki:

- 3-dimenzionalna matrika `matrix`.

Izhodni podatki:

- `data, int, poin` so podatki o matriki v CSR obliki zapisa;
- `pred` so podatki, vsebovani v `A[][][0]`.

3. `Sparse.FromCSRToNormal(int[] data, int[] ind, int[] poin,int[] pred)`
Metoda pretvori matriko, podano v CSR načinu, v standardni zapis matrike s 3-dimenzionalnim poljem.

Vhodni podatki:

- `data, int, poin` so podatki o matriki, podani v CSR načinu;
- `pred` so podatki o prednikih, ki jih moramo vključiti v matriko.

Izhodni podatki:

- 3-dimenzionalna matrika `C`, ki v `C[][][1]` vsebuje elemente matrike, v `C[][][0]` pa pripadajoče elemente iz tabele `pred`.

TESTNE MATRIKE

- **test18** je matrika dimenzije 200 x 200, s 4860 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.
- **test19** je matrika dimenzije 100 x 100, s 1863 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.
- **test20** je matrika dimenzije 300 x 300, z 9750 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.
- **test21** je matrika dimenzije 200 x 200, z 2128 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.
- **test22** je matrika dimenzije 100 x 100, s 177 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.

Vse matrike so sosedne matrike povezanih grafov.

REZULTATI

*Opomba: Časi v tabeli so zapisani v $10^6 ns$.

Tabelo moramo brati navpično, primer: pri matriki test18 je metoda AR delovala 36,90 10^6 ns in tako dalje.

OPAŽANJA

$MATRIKA$	$test18$	$test19$	$test20$	$test21$	$test22$
AR	36,90	5,12	119,54	37,43	5,06
CSR	176,90	21,75	638,28	170,54	16,00

Tabela 6.7: Rezultati testiranj Floyd-Warshallovega algoritma

Metoda, ki uporablja za zapis matrike CSR zapis, je veliko slabši v večini primerov. Definitivno se nam bolj splača uporabiti standardni način zapisa matrike. Problem zapisa CSR je, da pri množenju 2 matrik potrebujemo direktni dostop do indeksov elementov, ki pa jih pri tem zapisu nimamo na voljo. Zaradi tega je potrebno veliko pretvorb iz enega zapisa v drugega, da ugotovimo indekse elementov matrike in izvedemo množenje dveh matrik. Vse to pa seveda traja veliko dlje kot pri standardnem zapisu matrike, ko imamo te podatke vedno neposredno na voljo.

6.7 Primov algoritem

OPIS TESTA

Preverjali bomo zmogljivost Primovega algoritma. Implemetirali bomo algebrski Primov algoritem, in sicer z dvema metodama, od katerih vsaka uporablja drug način zapisa matrike. Za vsakega od teh zapisov smo implementirali metodo, ki po Primovem algoritmu vrne ceno minimalnega vpetega drevesa, ter metodo, ki po Primovem algoritmu vrne minimalno vpeto drevo samo. Torej imamo skupaj 4 metode, dve po dve uporabljata enak zapis matrike in dve po dve računata ceno drevesa ali pa drevo samo:

- standardni zapis s poljem (ARc) - metoda `Matrix.Primov_normal`
- standardni zapis s poljem (ARp) - metoda `Matrix.PrimovTree_normal_path`
- CSR zapis redke matrike (CSRc) - metoda `Sparse.Primov_CSR`
- CSR zapis redke matrike (CSRp) - metoda `Sparse.Primov_CSR_path`

Metode bomo pognali 10000-krat in izračunali povprečni čas, ki je zapisan v tabeli. 6.8.

OPIS UPORABLJENIH METOD

`Matrix.Primov_normal(int[][] matrika)`

Metoda simulira Primov algoritem, ki vrne ceno minimalnega vpetega drevesa, ne pa tudi minimalnega vpetega drevesa. Metoda je implementirana algoritmu 5.5.31 in uporablja standardni zapis matrike s poljem.

Vhodni podatki:

- matrika cene grafa `matrika`, za katerega nas zanima cena minimalnega vpetega drevesa.

Izhodni podatki:

- cena minimalnega vpetega drevesa.

Pomožne metode:

`1. Matrix.SestejVector(int[] v1,int[] v2)`

Metoda sešteje 2 vektorja. Metoda je implementirana po opisu seštevanja dveh vektorjev v poglavju 4.

Vhodni podatki:

- vektor `v1` dolžine n;
- vektor `v2` dolžine n.

Izhodni podatki:

- vektor `v` dolžine n, za katerega velja `v=v1+v2`.

`2. Matrix.argMin(int[] v)`

Metoda ugotavlja, do katerega vozlišča gre trenutno minimalna povezava od drevesa do vseh ostalih vozlišč, ki še niso v drevesu.

Vhodni podatki:

- vektor `v` dolžine n.

Izhodni podatki:

- vrednost minimalnega elementa v vektorju;

- indeks tega elementa v vektorju.

3. `Matrix.MinVector(int[] v1, int[] v2)`
Metoda združi vektorja `v1` in `v2` v skupni minimalni vektor v, za katerega velja `v[i]=min{v1[i],v2[i]}`.
Vhodni podatki:

- vektor `v1` dolžine n;
- vektor `v2` dolžine n.

Izhodni podatki:

- vektor `v` dolžine n, za katerega velja: `v[i]=min{v1[i],v2[i]}`.

`Matrix.PrimovTree_normal_path(int[][] matrika)`
Metoda simulira Primov algoritem, ki vrne minimalno vpeto drevo (v bistvu vrne katere povezave so v minimalnem vpetem drevesu). Metoda je implementirana po algoritmu 5.5.33 in uporablja standarni zapis matrike s poljem.
Vhodni podatki:

- matrika cene grafa `matrika`, za katerega nas zanima kakšno je minimalno vpeto drevo.

Izhodni podatki:

- vektor, ki vsebuje začetke in konce povezav, ki so vsebovane v minimalnem vpetem drevesu.

Pomožne metode:

1. `Matrix.SestejVector(int[] v1,int[] v2)`
2. `Matrix.argMin(int[] v)`
3. `Matrix.MinVector_path(int[] v1, int[] v2,int[] v1p, int[] v2p)`
Metoda vrne vsoto dveh vektorjev `v1` in `v2` tako, da na vsakem mestu vzame tistega, ki je manjši (tako da velja `rez[i]=min{v1[i],v2[i]}`), in pripadajočega prednika tega manjšega elementa.
Vhodni podatki:

- `v1` je prvi vektor, `v1p` so predhodniki povezav, katerih cene vsebuje ta vektor;
- `v2` je drugi vektor, `v2p` so predhodniki povezav, katerih cene vsebuje ta vektor.

Izhodni podatki:

- `LinkedList`, ki vsebuje: vektor `rez`, za katerega velja `rez[i]=min{v1[i],v2[i]}`, in vektor `pred`, ki vsebuje predhodnike povezav, ki so v `rez`.

`Sparse.Primov_CSR(int[]data,int[] ind, int[] poin)`
Metoda simulira Primov algoritem, ki vrne ceno minimalnega vpetega drevesa. Metoda je implementirana po algoritmu 5.5.31 in uporablja CSR zapis matrike.
Vhodni podatki:

- `data, ind, poin` so podatki o matriki, v CSR zapisu, ki je matrika cene grafa, za katerega nas zanima cena minimalnega vpetega drevesa.

Izhodni podatki:

- cena minimalnega vpetega drevesa grafa.

Pomožne metode
`1. Sparse.getVector(int[]data,int[] ind, int[] poin,int vrstica)`
Metoda vrne vektor, ki je podana vrstica matrike.
Vhodni podatki:

- `data, ind, poin` so podatki o matriki, podani v CSR načinu zapisa;
- `vrstica` je indeks vrstice, katero želimo "izločiti" iz matrike.

Izhodni podatki:

- vektor, ki je vrstica (indeksa vrstica) matrike.

`2. Matrix.argMin(int[] v)`
`3. Matrix.MinVector(int[] v1, int[] v2)`
`Sparse.Primov_CSR_path(int[]data,int[] ind, int[] poin)`

Metoda simulira Primov algoritem, ki vrne minimalno vpeto drevo (v bistvu vrne katere povezave so v minimalnem vpetem drevesu). Metoda je implementirana po algoritmu 5.5.33 in uporablja CSR zapis matrike.

Vhodni podatki:

- `data, ind, poin` so podatki o matriki, v CSR zapisu, ki je matrika cene grafa, za katerega nas zanima cena minimalnega vpetega drevesa.

Izhodni podatki:

- vektor, ki vsebuje začetke in konce povezav, ki so vsebovane v minimalnem vpetem. drevesu

Pomožne metode:

```
1. Sparse.getVector(int[]data,int[] ind, int[] poin,int vrstica)
2. Matrix.argMin(int[] v)
3. Matrix.MinVector_path(int[] v1, int[] v2,int[] v1p, int[] v2p)
```

TESTNE MATRIKE

- **test18** je matrika dimenzije 200 x 200, s 4860 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.
- **test19** je matrika dimenzije 100 x 100, s 1863 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.
- **test20** je matrika dimenzije 300 x 300, z 9750 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.
- **test21** je matrika dimenzije 200 x 200, z 2128 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.
- **test22** je matrika dimenzije 100 x 100, s 177 vrednostmi od 1 do 5, po diagonali so ničle, drugje pa ∞.

Vse matrike so sosedne matrike povezanih grafov.

REZULTATI

$MATRIKA$	$test18$	$test19$	$test20$	$test21$	$test22$
ARc	22,94	5,95	53,27	22,58	5,76
ARp	51,86	13,06	140,41	51,40	12,63
CSRc	45,14	10,88	103,56	44,98	10,93
CSRp	69,51	16,91	178,29	69,60	16,67

Tabela 6.8: Rezultati testiranj Primovega algoritma

*Opomba: Časi v tabeli so zapisani v $10^4 ns$.
Tabelo moramo brati navpično, primer: pri matriki test18 je metoda ARc delovala 22,94 10^4 ns in tako dalje.

OPAŽANJA

Kot je bilo pričakovate: metodi, ki računata tudi pot in ne samo cene, potrebujeta več časa. Zmogljivost tako metode, ki uporablja standarden zapis matrike in tiste s CSR zapisom matrike, je odvisna samo od števila vozlišč. Kot vidimo pa je implementacija s standardnim zapisom manj kompleksna, ker imamo neposreden dostop do vrstice matrike, zato je tudi hitrejša. Opazimo, da nam pri tem algoritmu ne pomaga zapis redke matrike v načinu CSR, ker ne preiskujemo povezav, ampak vozlišča. Lastnosti vozlišč (lega v matriki) in njihovih povezav pa so hitreje dostopne s standardnim zapisom matrike, zato je tudi ta zapis bolj primeren za uporabo pri Primovem algoritmu.

Poglavje 7

Zaključek

Skozi delo smo prišli do zaključka, da je CSR zapis redke matrike precej neprimeren za algoritme, kjer se število neničelnih elementov ves čas spreminja, saj nimamo neposrednega dostopa do vrstice elementa v matriki, zato je potrebno veliko pretvorb iz standardnega načina zapisa v CSR zapis, da ugotovimo pravo zaporedje podatkov v data. Prav tako je CSR zapis precej slabši od standardnega zapisa matrike, kadar moramo množiti dve matriki, saj tudi tu potrebujemo podatke o vrstici, v kateri je element; ta podatek pa je iz CSR zapisa potrebno preračunati, medtem ko je v standardnem zapisu neposredno dostopen, to pa nam jemlje čas. CSR zapis je predvsem neprimeren, kadar je iz matrike, zapisane v CSR zapisu, potrebno dobiti vse elemente kateregakoli stolpca, saj ti podatki niso direktno dostopni in je potrebno pregledati vse podatke, da ugotovimo, kateri so ti elementi za vsak stolpec. Problem sprehoda bi se seveda dalo rešiti tako, da bi uporabili CSC zapis, kjer elemente po vrsti shranjujemo po stolpcih, a tu bi imeli podobne težave, ko bi potrebovali elemente vrstice. V večini naših algoritmov potrebujemo oboje (stolpce in vrstice) zaradi množenja 2 matrik, zato zapis CSC ne bi dal bistveno drugačnih rezultatov kot CSR.

CSR zapis je primeren, kadar množimo matriko z vektorjem, in v redkih primerih, ko potrebujemo množenje matrik v kolobarju (+,*); takoj ko kolobar spremenimo, naletimo na težave.

Metode bi se dalo še časovno optimizirati na primer s paralelnim programiranjem,

ampak bi to lahko narediti tako pri metodah, ki uporabljajo zapise redkih matrik, kot pri metodah, ki uporabljajo standardni zapis matrike z dvodimenzionalno tabelo, zato se primerjava v zmogljivostih ne bi bistveno spremenila.

Nadaljnje delo

V delu smo preučili možnosti implementacije nekaj algoritmov na grafu z matrikami, ki smo jih zapisali na standarden način s poljem in z nekaj različnimi enostavnimi zapisi redkih matrik. Ugotovili smo, da ti enostavni zapisi redkih matrik v večini primerov, ki smo jih mi preučili, ne morejo konkurirati s standardnim zapisom matrike. Verjamemo, da bi s kompleksnejšimi zapisi redkih matrik (npr. v delu smo omenili tudi Ellpack-Itpackov zapis, obstaja pa še veliko drugih) lahko dosegli boljše čase in bi v več primerih presegli čase metod, ki uporabljajo standardni zapis matrik.

Drugi pogled, ki bi ga na to delo lahko navezali, je primerjava standardne implementacije opisanih algoritmov na grafih z algebrsko implementacijo, ki uporablja izračune na sosednih matrikah. Tudi to je zelo zanimiva tema, katere se je že lotilo kar nekaj raziskovalnih skupin,ki so tudi dokazale, da so izračuni na sosednih matrikah lahko hitrejši kot izračuni na grafih, ravno zato so že sedaj na primer v MATLABU matrike postale najpogosteje uporabljena struktura za reševanje problemov na grafih.

Literatura

[1] Jack J. Dongarra and others, "Graph Algorithms in the Language of Linear Algebra", str 1-58, 2011.

[2] Seymour Lipschutz, Ph.D., Marc Lipson, Ph.D, "Linear algebra (Third Edition)", str 1-41, 2001.

[3] Ivan Vidav, "Algebra", str 84-92, 123-137, 2003.

[4] Tomaž Dobravec,"Algoritmi nad grafi v jeziku linearne algebre", LALGinar, FRI 2012.

[5] Yousef Saad, "Iterative Methods for Sparse Linear Systems", str 68-95, 2000.

[6] Metode za redke matrike , dosegljivo na http://netlib.org/utk/papers/templates/node90.html

[7] MSR format za redke matrike, dosegljivo na http://www.iue.tuwien.ac.at/phd/wagner/ node83.html

[8] Teorija grafov, dosegljivo na http://164.8.132.54/GIS/sesto.html

[9] Definicije, dosegljivo na http://sl.wikipedia.org/wiki

[10] Redke matrike, dosegljivo na http://www.answers.com/topic/sparse-matrix

[11] Algoritmi, dosegljivo na http://wiki.fmf.uni-lj.si/wiki

[12] Uteženi graf (omrežje), dosegljivo na http://www.nauk.si/materials/5603/out/#state=1

[13] Zapisi redkih matrik, dosegljivo na http://netlib.org/utk/papers/templates/node98.html

[14] Graf in algoritmi na grafu, dosegljivo na http://rtk.ijs.si/misc/Grafi.ppt

Slike

3.1 Primer dualnosti med grafom in matriko 13
3.2 Primer matrike sosednosti za 4 točke 13
3.3 Primer matrike sosednosti za več točk 14
3.4 Primer CSR zapisa na matriki A 17
3.5 Primer CSC zapisa na matriki A 17
3.6 Primer zapisa s terkami na matriki A 18
3.7 Primer MSR zapisa na matriki A 19
3.8 Primer shranjevanja diagonalne matrike na matriki B 20
3.9 Primer shranjevanja diagonalne matrike z Ellpach-Itpackovim zapisom na matriki B 21

5.1 Slika nivojev pri preiskovanju v širino 32
5.2 Primer računanja v kolobarju (ALI,IN) 32
5.3 Simulacija BFS algoritma 33
5.4 Simulacija BFS algoritma na primeru 33
5.5 Primer množenja v kolobarju (min,+) 36
5.6 Primer iskanja minimalnih poti 38
5.7 Primer izračuna po Bellman-Fordovem algoritmu 45
5.8 Primer rekonstrukcije poti po Floyd-Warshallovem algoritmu 56
5.9 Skica delovanja Primovega algoritma 58
5.10 Računanje cene minimalnega vpetega drevesa 61

Tabele

2.1 Tabela algebrskih struktur in zapisov 9

6.1 Rezultati testiranj algoritmov za množenje matrike z vektorjem . . 67

6.2 Rezultati testiranj algoritma BFS 72

6.3 Rezultati testiranj algoritma iskanja minimalnih poti 76

6.4 Rezultati testiranj Bellman-Fordovega algoritma brez shranjevanja poti 81

6.5 Rezultati testiranj Bellman-Fordovega algoritma na povezanih grafih 83

6.6 Rezultati testiranj Bellman-Fordovega algoritma, ki vrača tudi minimalno pot 84

6.7 Rezultati testiranj Floyd-Warshallovega algoritma 88

6.8 Rezultati testiranj Primovega algoritma 93